А.Мешков, Ю.Тихомиров

VISUAL C++ И MFC.

ПРОГРАММИРОВАНИЕ ДЛЯ WINDOWS NT И WINDOWS 95

Том 2

Содержание

Введение 1

ЧАСТЬ VII. ГРАФИЧЕСКИЙ ИНТЕРФЕЙС УСТРОЙСТВ (GDI) 5

Глава 16. Классы графического интерфейса 7

Контексты устройств. 8

Графические объекты 10

Глава 17. Класс CDC 14

Создание объекта класса и его связь с контекстом устройства 14

Данные класса 15

Конструктор 16

Инициализация 16

Доступ к объектам рисования 18

Функции контекста устройства 18

Настройка процесса рисования 18

Функции средств рисования 19

Установка объектов рисования 19

Функции настройки цветов и палитр 21

Режимы рисования 21

Режим отображения 24

Настройка режимов отображения 24

Преобразование координат 30

Функции рисования 30

Отображение регионов 30

Отсечения 30

Рисование линий 30

Базовые функции рисования 31

Отображение эллипсов и многоугольников 33

Контуры 33

Отображение битовых массивов 36

Функции, использующие битовые массивы 36

Прокрутка 39

Управление выводом текста 39

Вывод текста 39

Информация о шрифтах 45

Дополнительные функции 46

Интерфейс низкого уровня с устройствами 46

Управление процессом печати документов 46

Метафайлы 46

Глава 18. Классы графических объектов 47

Класс CGdiObject 47
Данные класса 47
Функции-члены класса 47
Карандаш (класс CPen) 52
Кисть (класс CBrush) 58
Шрифт (класс CFont) 59
Битовый массив (класс CBitmap) 66
Регион (класс CRgn) 74
Создание регионов 74
Комбинирование регионов 76
Глава 19. Пример приложения PQClock 79
Класс приложения 79
Класс окна 82
ЧАСТЬ VIII. СТАНДАРТНЫЕ ЭЛЕМЕНТЫ УПРАВЛЕНИЯ (CONTROLS) 95
Глава 20. Обзор стандартных элементов управления 97
Создание элементов управления 97
Использование редактора ресурсов для создания элементов управления 98
Создание элементов управления в тексте программы 98
Глава 21. Классы стандартных элементов управления 100
Статические элементы управления (класс CStatic) 100
Конструктор 100
Инициализация 100
Вывод битового массива 103
Вывод пиктограммы 104
Вывод курсора 105
Вывод расширенных метафайлов 105
Обработка сообщения WM_CTLCOLOR 106
Кнопки (класс CButton) 108
Инициализация 111
Функции для определения и изменения текущего состояния кнопок 112
Функции для определения и изменения стиля кнопки 113
Назначение изображений 113
Переопределяемые функции 114
Простейший текстовый редактор (класс CEdit) 115
Инициализация 116
Выделение текста 118
Работа со строками 119
Работа с блоками теста 121
Параметры и флаги 122
Функции только для Windows 95 123
Список (класс CListBox) 124
Инициализация 125
Изменение параметров списка 127

Функции для работы с содержимым списка 129
Операции над элементами списка 132
Переопределяемые функции 134
Пример самоотображаемого списка 137
Комбинированный список (класс CComboBox) 141
Инициализация 142
Основные операции с комбинированным списком 144
Операции со строками и виртуальные функции 147
Полоса прокрутки (класс CScrollBar) 148
Создание объекта 151
Функции для работы с полосой прокрутки 153
Глава 22. Классы с расширенными возможностями 157
Кнопки с нанесенным битовым изображением (класс CBitmapButton) 157
Кнопки в блоке диалога 158
Кнопки, созданные программным путем 159
Функции-члены класса 159
Список, имеющий кнопки выбора (класс CCheckListBox) 161
Инициализация 161
Функции для работы со списком 161
Виртуальные функции 162
Список с возможностью перемещения элементов (класс CDragListBox) 170
Ограничения на использование 170
Создание списка 170
Функции, предназначенные для перемещения элементов 171
Методика обхода ограничений 172
Реализация копирования элементов списка 172
ЧАСТЬ IX. ОБЩИЕ ЭЛЕМЕНТЫ УПРАВЛЕНИЯ (COMMON CONTROLS) 177
Глава 23. Использование общих элементов управления 179
Обзор 179
Общие элементы управления в блоках диалога 180
Общие элементы управления в качестве дочерних окон 181
Уведомления от общих элементов управления 181
Глава 24. Вспомогательные элементы управления 183
Просмотр видео-клипов (класс CAnimateCtrl) 183
Применение элемента анимации 184
Использование класса CAnimateCtrl 185
Уведомления 186
Создание видео-клипов 188
Клавиша вызова (класс CHotKeyCtrl) 189
Глобальные клавиши вызова 192
Поточно-ориентированные клавиши вызова 192
Пример использования класса CHotKeyCtrl 193
Индикатор (класс CProgressCtrl) 199

Использование индикатора в панели диалога 201
Линейка с бегунком (класс CSliderCtrl) 205
Уведомления 205
Создание элемента управления 206
Параметры элемента управления 207
Положение меток 208
Положение бегунка 209
Спин (класс CSpinButtonCtrl) 209
Уведомления 210
Автоматическое изменение 212
Создание спина 212
Параметры элемента управления 213
Всплывающая подсказка (класс CToolTipCtrl) 215
Оконная поддержка 216
Панели инструментов с подсказками 218
Возвращаемся к классу CToolTipCtrl 219
Глава 25 Комплексные элементы управления 223
Список изображений (класс CImageList) 223
Создание списка 224
Изменение содержимого списка 225
Рисование изображений 226
Перемещение изображений 227
Информация о списке 231
Заголовок (класс CHeaderCtrl) 231
Создание элемента заголовка 232
Изменение элементов заголовка 234
Обработка извещений элемента заголовка 235
Элемент заголовка, внешний вид которого определяется пользователем 238
Просмотр списка (класс CListCtrl) 239
Режимы вывода 239
Создание просмотра списка 240
Параметры просмотра списка 242
Функции для работы со списком в целом 242
Списки изображений 244
Основные и дополнительные поля 245
Записи по запросу (Callback Items), 245
Изменение содержимого просмотра списка 246
Поиск и сортировка записей 250
Работа со столбцами 254
Редактирование надписей записей 255
Обработка уведомлений 256
Реализация просмотра списка с возможностью перемещения записей 257
Переопределяемые функции 258

ListCtrl — пример приложения 258
Просмотр дерева (класс CTreeCtrl) 266
Создание просмотра дерева 267
Списки изображений 268
Функции для работы с просмотром дерева в целом 268
Изменения содержимого дерева 269
Поиск и сортировка записей дерева 275
Обработка уведомлений 279
ЧАСТЬ X. АРХИТЕКТУРА "ДОКУМЕНТ/ПРЕДСТАВЛЕНИЕ" 281
Глава 26. Основные положения **283**
Место объекта-приложения в архитектуре "документ/представление" 286
Класс CDocTemplate 286
Класс CSingleDocTemplate 291
Класс CMultiDocTemplate 292
Роль фреймов в архитектуре "документ/представление" 300
Глава 27. Документ и его представления **306**
Документы 307
Класс CDocument 309
Класс CArchive 317
Представления 320
Класс CView 323
Класс CCtrlView 327
Класс CEdit View 328
Классы CListView и CTreeView 334
Класс CScrollView 334
Класс CSplitterWnd 339
Глава 28. Печать и предварительный просмотр документов **349**
Выбор и настройка параметров принтера 351
Создание контекста устройства 354
Печать документов и библиотека MFC 354
Предварительный просмотр документа 364
ЧАСТЬ XI. ЭЛЕМЕНТЫ ИНТЕРФЕЙСА ПОЛЬЗОВАТЕЛЯ 373
Глава 29. Меню **375**
Основные типы меню 375
Класс CMenu 378
Создание меню на основе шаблона 386
Динамическое создание меню (приложение DynMenu) 391
Динамическое изменение меню 393
Системное меню 399
Контекстное меню 402
Создание собственных маркеров состояния 405
Самоотображение элементов меню 406
Глава 30. Акселераторы, графические и другие ресурсы **408**

Акселераторы 408
Графические ресурсы 410
Строковые ресурсы 411
Ресурсы, определяемые программистом 412
ПРИЛОЖЕНИЯ 413
Приложение 1. Иерархия классов MFC 4.1 415
Приложение 2. Функции-члены класса CDC 418
Конструктор 418
Инициализация 418
Работа с контекстом устройства 419
Объекты рисования 420
Работа с цветом и цветовыми палитрами 420
Настройка параметров рисования 421
Настройка режимов отображения 421
Преобразование координат 423
Работа с регионами 423
Работа с областью отсечения 424
Рисование линий 425
Простейшее рисование изображений 426
Рисование эллипсов и многоугольников 427
Рисование изображений 428
Отображение текстовых строк 430
Работа со шрифтами 432
Управление устройствами 433
Прокрутка 434
Работа с метафайлами 434
Работа с контурами 434
Приложение 3. Значения параметра функции CDC::GetDeviceCaps 436
Предметный указатель 441

Предметный указатель

A
ACN_START 186
ACN_STOP 186
ACS_AUTOPLAY 185
ACS_CENTER 185
ACS_TRANSPARENT 185
AfxRegisterWndClass 38
ALTERNATE 22
AnsiToOem 117, 143
ANSI_CHARSET45,61
ANSI_FtXED_FONT 20
ANSI_VAR_FONT 20
ARABIC_CHARSET 45, 61
B
BALTIC_CHARSET 45, 61
BLACK_BRUSH 20
BLACK_PEN 20
BLACKNESS 36
BLACKONWHITE 23
BS_3STATE 110
BS_AUTO3STATE 110
BS_AUTOCHECKBOX 109
BS_AUTORADIOBUTTON 109
BS_BITMAP 110

BS_BOTTOM 110
BS_CENTER 110
BS_CHECKBOX 109
BS_DEFPUSHBUTTON 109
BS_FLAT 110
BS_GROUPBOX 109
BS_ICON 110
BS_LEFT110
BS_LEFTTEXT 110
BS_MULTILINE 110
BS_NOTIFY 110
BS_OWNERDRAW 110, 158
BS_PUSHBUTTON 109
BS_PUSHLIKE110
BS_RADIOBUTTON 109
BS_RIGHT 110
BS_RIGHTBUTTON 110
BS_TEXT 110
BS_TOP 110
BS_VCENTER 110
C
CAnimateCtrl
Close 185
Create 185
Open 185
Play 186
Seek 186
Stop 186
CArchive318
Abort 318
bNoFlushOnDelete318
Close 318
Flush 319
IsLoading 320
IsStoring 320
load 318
m_pDocument 317
Read 319
ReadString319
store 318
Write 319
WriteString 320
CBitmap
CreateBitmap 66, 71
CreateBitmapInderect 66
CreateCompatibleBitmap 20, 66, 70
CreateDiscardableBitmap 66
GetBitmap 69
GetBitmapBits 67
GetBitmapDimension 51, 67, 68
LoadBitmap 67
LoadMappedBitmap 67
LoadOEMBitmap 67
HBITMAP 51
SetBitmapBits 67, 71
SetBitmapDimension 51, 67, 68
CBitmapButton
AutoLoad 159, 160
LoadBitmaps 160
SizeToContent 160
CBN_CLOSEUP 142
CBN_EDITUPDATE 141
CBN_SELENDCANCEL 142
CBN_SELENDOK 142
CBrush 58
CreateBrushIndirect 58
CreateDIBPattemBrush 58
CreateHatchBrush 58, 93
CreatePattemBrush 58
CreateSolidBrush 58, 93
CreateSysColorBrush 58
HBRUSH51
CBS_AUTOHSCROLL 143
CBS_DISABLENOSCROLL 144
CBS_DROPDOWN 141, 143, 146
CBS_DROPDOWNLIST 141, 143, 146
CBS_HASSTRINGS 143
CBS_NOINTEGRALHEIGHT 144
CBS_OEMCONVERT 143
CBS_OWNERDRAWFIXED 143
CBS_OWNERDRAWVARIABLE 143
CBS_SIMPLE 141, 143
CBS_SORT 144

CButton
Create 111, 160
DrawItem 114
GetButtonStyle 113
GctCheck 112
GetState 112
SetBitmap 114
SetButtonStyle 113
SetCheck 113
SetCursor 114
Setlcon 114
SetState 112
CCheckListBox
Create 161
DrawItem 163
Enable 162
GetCheck 162
GetCheckStyle 162
IsEnabled 162
MeasureItem 163
OnGetCheckPosition 162
SetCheck 162
SetCheckStyle 162
CComboBox
AddString 135, 148
Clear 144
Copy 144
Create 142
Cut 144
DeleteString 148
Dirl48
DrawItem 148
FindString 148
FindStringExact 148
GetCount 145
GetCurSel 145
GetDroppedControlRect 146
GetDroppedState 146
GetDroppedWidth 146
GetEditSel 144
GetExtendedUI 147
GetHorizontalExtent 145
GctItemData 145
GetttemDataPtr 145
GetItemHeight 145
GetLBText 142, 147
GetLBTextLen 147
GetLocale 146
GetText 143
GetTopIndex 145
InitStorage 146
InsertString 135, 148
LimitText 144
MeasureItem 148
Paste 144
ResetContent 148
SelectString 148
SetCurSel 145
SetDroppedWidth 146
SetEditSel 146
SetExtendedUI 147
SetHorizontalExtent 145
SetItemData 145
SetItemDataPtr 145
SetItemHeight 145
SetLocale 146
SetTopIndex 145
ShowDropDown 146
CControlBar
SetBarStyle218
CCS_BOTTOM 233
CCS_NOPARENTALIGN 233
CCS_NORESIZE 233
CCS_TOP 233
CCtrlView
CCtrlView 328
m_dwDefaultStyle 327
m_strClass 327
CDC
AddMetaFileComment 46
AngleArc 30, 33
Arc 30,33,86
ArcTo 30, 33
BeginPath 33, 89
BitBIt 37

Chord 33, 53
CloseFigure 33
CreateCompatibleDC 16
CreateDC 16
DeleteTempMap 18
DPtoHl METRIC 30
DPtoLP 30
Draw3dRect31
DrawEdge 31
DrawEscape 46
DrawFocusRect 33
DrawFrameControl 31
Drawlcon 31
DrawState31,32, 72
DrawText 41
Ellipse 33, 53, 86
EndDoc 46
EndPage 46
EndPath 33, 89
EnumObjects 19
Escape 46
ExcludeClipRect 30
ExcludeUpdateRgn 30
ExtFloodFill 39, 87
ExtTextOut 24, 33, 40
FillPath33,89
FillRect31,92
FillRgn 30
FillSolidRect31
FloodFill 39
FrameRect 31
FrameRgn 30
FromHandle 17
GetBrushOrg 19
GetCharABCWidth 45
GetCharWidth 45
GetClipBox 30
GetCurrentBitmap 18
GetCurrentBrush 18
GetCurrentFont 18
GetCurrentPalette 18
GetCurrentPen 18
GetDeviceCaps 17, 18
GetFontData 45
GetGlyphOutline 45
GetHalftoneBrush 21
GetKerningPairs 45
GetOutlineTextMetrics 45
GetOutputCharWidth 45
GetOutputTextExtent 40
GetOutputText Metrics 43
GetPixel 36 GetSafeHdc 18
GetTextAllign 40
GetTextExtent 40
GetTextFace 40
GetTextMetrics 43, 90
HIMETRICtoDP 30
HIMETRICtoLP 30
IntersectClipRect 30
InvertRect 31
InvertRgn 30
LineTo 30, 33, 53, 87
LPtoDP 30, 87
LPToHIMETRIC 30
m_hAttribDC 15
m_hDC 15
MoveTo 33, 87
PaintRgn 30, 88
PatBIt 36
Pie 33
PlayMetaFile 46
PlgBIt 36
PolyBezier 30, 33, 89
PolyBezierTo 30, 33
PolyDraw 30, 33
Polygon 33, 91
Polyline 30, 33
PolvlineTo 30, 33
PolyPolygon 33
PolyPolyline 30, 33
PtVisible 30
Rectangle 33, 53
RestoreDC 18
RoundRect 33, 53, 86
SaveDC 18
ScrollDC 39

SelectClipPath 35
SelectClipRgn 20, 30, 35, 87
SelectObject 19, 86
SelectPalette21
SelectStockObject 20, 50
SetAttribDC 15
SetBkColor 24, 89
SetBkMode21,90
SetBrushOrg 19
SetMapMode 25, 85
SetMiterLimit 55
SetOutputDC 15
SetPixel 36
SetPixelV 36
SetPolyFillMode21
SetROP2 22
SetStretchBltMode 23
SetTextAlign40,41,43,89
SetTextColor 24
SetTextJustification 40
SetViewportExt 28, 85
SetViewportOrg 26, 85
SetWindowExt 28, 85
SetWindowOrg 26
StartDoc 46
Start Page 46
StokePath 33
StretchBIt 38
StretchBltMode 38
StrokeAndFillPath 33
TabbedTextOut 40
TextOut 24, 33, 40, 90
UpdateColors21

CDialog
DoModal 181
OnInitDialog 181
OnOK 181

CDocManager
m_templateList 296

CDocTemplate 287
AddDocument 287
CloseAllDocuments 289
CreateNewDocument 290
CreateNewFrame 290
docName 288
fileNewName 288
filterExt 289
filterNewName 289
GeNextDoc 292
GetDocString 288
GetFirstDocPosition 292
InitialUpdateFrame 291
LoadTemplate 287
MatchDocType 290
OpenDocumentFile 292
regFileTypeId 289
regFileTypeName 289
RemoveDocument 287
SaveAIIModified 289
SetDefaultTitle 292
windowTitle 288

CDocument 309
Add View 310
CanCloseFrame 312
DeleteContents312
GetDocTemplate 309
GetFile314
GetFirstViewPosition 310
GetNextView310
GetPathName310
GetTitle310
IsModified310
OnChangedViewList 313
OnCloseDocument 313
OnNewDocument 313
OnOpenDocument 313
OnSaveDocument 313
PreCloseFrame314
ReleaseFile314
RemoveView 310
ReportSaveLoadException 3!4
SaveModified 314
SetModifiedFlag310
SetPathName310
SetTitle310
UpdateAllViews311

CDragListBox
BeginDrag 171
CancelDrag 171
Dragging 172
Drawlnsert 171
DrawSingle 171
Dropped 172
ItemFromPt 171
m_nLast 171
CEdit
CanUndo 121, 122
CharFromPos 124
Clear 121
Copy 121
Create 116
Cut 121
EmptyUndoBuffer 122
GetFirstVisibleLine 119
GetHandle 121
GetLimitText 124
GetLine 120
GetLineCount 120
GetMargms 123
GetModify 121, 122
GetPasswordChar 123
GetRect 123
GetSel 118
LimitText 123
LineFromChar 120
Linelndex 120
LineLength 120
LineScroll 119
Paste 121
PosFromChar 124
ReplaceSel 119
SetHandle 121, 122
SetLimitText 123, 124
SetMargins 124
SetModify 122
SetPasswordChar 123
SetReadOnly 123
SetRect 123
SetRectNP 123
SetSel 118
Undo 121, 122
CEditView 329
dwStyleDefault 329
FindText 330
GetButTerLength 330
GetPrinterFont 329
GetSelectedText 330
LockBuffer 330
OnFindText 330
OnReplaceAll 331
OnReplaceSel 331
OnTextNotFound331
PrintInsideRect 329
SerializeRaw 330
SetPrinterFont 329
SetTabStops 329
UnlockBuffer 330
CF_TEXT 121, 144
CFont
CreateFont 59, 64, 93
CreateFontIndirect 59
CreatePointFont 59, 65
CreatePointFontIndirect 59, 65
HFONT 51
CFrameWnd
LoadAccelTable 409
OnEndPrintPreview 369
OnSetPreviewMode 369
CGdiObject
Attach 50, 67
CreateStockObject 50, 58
DeleteObject 48, 88
FromHandle 48
GetObject 51,69
GetSafeHandle 47
m_h0bject 47
HGDIOBJ 51
CHeaderCtrl
Create 233
DeleteItem 235
DrawItem 238
GetItem 235

GetItemCount 235
InsertItem 235
SetItem 235
CHINESEBIG5_CHARSET 45, 61
CHotKeyCtrl
Create 189
GetHotKey 190, 191, 193
SetHotKey 190
SetRules 191
CImageList
Add 225
Attach 224, 226
BeginDrag 227
Create 224, 225
Detach 224, 226
DragEnter 227
DragLeave 228
DragMove 227
DragShowNolock 228
Draw 226
EndDrag 228
Extractlcon 225
Getlmagelnfo 231
GetSafeHandle 224
m_hlmageList 224
Read 226
Remove 225
Replace 225
SetBkColor 226
SetDragCursorImage 228
SetOverlavImage 227
Write 226
CLIP_CHARACTER_PRECIS 63
CLIP_DEFAULT_PRECIS 63
CLIP_EMBEDDED 63
CLIP_LH_ANGLES 63
CLIP_MASK 63
CLIP_STROKE_PRECIS 63
CLIP_TT_ALWAYS 63
CListBox
AddStringl29, 131, 135
CompareItem 130, 136
Create 125
DeleteString 131
Dir 130,131
DrawItem 134, 137
FindString 131
FindStringExact 132
GetCount 132
GetCurSel 132
GetHorizontalExtent 127
GetItemData 133
GetItemDataPtr 133
GetItemHeight 128
GetItemRect 127
GetLocale 129 GetSel 132
GetSelItems 132, 133
GetText 133
GetTextLen 133
GetTopIndex 133
InitStorage 131
InsertString 130, 131, 135
ItemFromPoint 134
MeasureItem 135, 137
ResetContent 131
SelectString 131
SetColumnWidth 125, 127, 128
SetCurSel 125, 132
SetHorizontalExtent 127
SetItemData 133
SetItemDataPtr 133
SetItemHeight 128
SetItemRange 133
SetLocale 129
SetSel 132
SetTabStops 128
SetTopIndex 133
CListCtrl
Arrange 243 Create 240
CreateDragImage 258
DeleteAllItems 246
DeleteColumn 255
DeleteItem 248
DrawItem 258
EditLabel 256
EnsureVisibte 243

FindItem 251
GetBkColor 242
GetCallbackMask 246
GetColumn 255
GetColumnWidth 255
GetCountPerPage 243
GetEditControl 256
GetImageList 244
GetItem 248
GetItemCount 246
GetItemData 249
GetItemPosition 250
GetItemRect 250
GetItemState 249
GetItemText 249
GetNextItem 250
GetOrigin 243
GctSelectedCount 246
GetStringWidth 254
GetTextBkColor 242
GetTextColor 242
GetTopIndex 242
GetViewRect 242
HitTest 252
InsertColumn 254
InsertItem 247
RedrawItems 243
Scroll 243
SetBkColor 242
SelCallbackMask 246
SctColumn 255
SetColumnWidth 255
SetImageList 244
SetItem 248
SetItemCount 246
SetItemData 249
SetItemPosition 250
SetItemState 249
SetIlemText 249
SetTextBkColor 242
SetTextColor 242
SortItems 253
Update 243

CListView
CListView 334
GetListCtrl 334
CLRJSONE 226, 242
CMenu 378
AppendMenu 135, 380
CheckMenuItem 383
CheckMenuRadioItem 383
CreateMenu 378
CreatePopupMenu 378
DeleteMenu 382
DestroyMenu 380
DrawItem 385
EnableMenuItem 382
GetMenuContextHelpId 386
GetMenuItemCount 383
GetMenuItemID 384
GetMenuState 384
GetMenuString384
GetSubMenu 384
InsertMenu 135, 381
LoadMenu 378
LoadMenuIndirect 379
m_hMenu 378
MeasureItem 385
ModifyMenu 135,382
RemoveMenu 382
SetMenuConiextHelpId 386
SetMenuIteinBitmaps 383
TrackPopupMenu 384
CMultiDocTemplate
CMultiDocTemplate 293
COLORONCOLOR 23
COMPAREITEMSTRUCT 137
CPalette
HPALETTE51 **CPen** 52
CreatePen 58, 93
CreatePenIndirect 58
HPEN 51
CProgressCtrl
Create 200
Off set Pos 201

SetPos 200
SetRange 200
SetStep 201
Stepit 200
CreateDIBitmap72
CRgn
CombineRgn 76, 88
CopyRgn 77
CreateEllipticRgn 75
CreateEllipticRgnIndirect 75, 88
CreateFromPath 77
CreatePolygonRgn 75
CreatePolyPolygonRgn 75
CreateRectRgn 74
CreateRectRgnIndirect 74, 87
CreateRoundRectRgn 75
EqualRgn 77
HRGN51
CRichEditCtrl 180
CScrollBar
Create 151
EnableScrollBar 154
GetScrollInfo 155
GetScrollLimit 156
GetScrollPos 153
GetScrollRange 153
SetScrollBar 150
SetScrollInfo 155
SetScrollPos 154
SetScrollRange 153
ShowScrollBar 154
CScrollView 335
GetDeviceScrollPosition 337
GetDeviceScrollSizes 336
GetScrollPosition 337
GetTotalSize 336
ResizeParentToFit 339
ScrollToPosition 337
SetScaleToFitSize 337
SetScrollSizes 335
CSingleDocTemplate 291
CSliderCtrl
ClearSel 208
ClearTics 209
Create 206
GetChannelRect 208
GetLineSize 208
GetNumTics 208
GetPageSize 208
GetPos 209
GetRange 208
GetRangeMax 208
GetRangeMin 208
GetSelection 208
GetThumbRect 208
GetTic 208
GetTicArray 208
SetLineSize 208
SetPageSize 208
SetPos 209
SetRange 208
SetRangeMax 208
SetRangeMin 208
SetSelection 208
SetTicFreq 209
VerifyPos 209
CSpinButtonCtrl
Create 212
GetBase215
GetBuddy 213
GetPos 213
GetRange 214
SetAccel 214
SetBase215
SetBuddv 213
SetPos 213
SetRange 214
CSplitterWnd 342
ActivateNext 348
CanActivateNext 348
Create 342
CreateScrollBarCtrl 348
CreateStatic 343
CreateView 344
DeleteColumn 347

DeleteRow 347
DeleteView 347
DoKeyboardSplit 348
DoScroll 348
DoScrollBy 348
GetActivePane 347
GetColumnCount 345
GetColumnInfo 345
GetPane 345
GetRowCount 345
GetRowInfo 345
GetScrollStyle 346
IdFromEowCol 346
IsChildPane 346
OnDrawSplitter 346
On Invert Tracker 347
RecalcLayout 346
SetActivePane 347
SetColumnInfo 345
SetRowInfo 345
SetScrollStyle 346
SplitRow 347
CSroIlBar
SetScrollRange 156
CStatic 100
Create 101
GetBitmap 103
GetCursor 105
GetEnhMetaFile 105
Getlcon [04
SetBitmap 103
SetCursor 105
SetEnhMetaFile 105
Setlcon 105
CString
LoadString 411
CToolBar
Create 218
CToolBarCtrl
GetToolTips218
SetToolTips218
CToolTip
Create 220
CToolTipCtrl
Activate 221
AddTool 220
DelTool 221
GetText 222
GetToolInfo 222
HitTest 222
RelayEvent 222
SetDelayTime 222
SetToolInfo 222
UpdateTipText 222
CTreeCtrl
Create 267
CreateDragImage 280
DeleteAllItems 269
DeleteItem 272
EditLabel 280
EnsureVisible 276
Expand 275
GetChildItem 274
GetCount 269
GetDropHilightItem 277
GetEditControl 280
GetFirstVisibleItem 277
GetImageList 268
GetIndent 268
GetItem 273
GetItemData 274
GetItemlmage 273
GetItemRect 274
GetItemState 273
GetItemText 273
GetNextItem 276
GetNextSiblingItem 276
GetNextVisibleItem 277
GetParentItem 276
GctPrevSiblingItem 276
GetPrevVisibleItem 277
GetRootItem 277
Get Selected Item 277
GetVisibleCount 269
HitTest 277
InsertItem 269

ItemHasChildren 274
Select 275
SelectDropTarget 275
SelectItem 275
SetImageList 268
SetIndent 268
SetItem 272
SetItemData 274
SetItemImage 273
SetItemState 273
SetItemText 274
SortChildren 278
SortChildrenCB 278
CTreeView 334
GetTreeCtrl 334
CView 323
DoPreparePrinting 357
DoPrintPreview 368
GetDocument 323
OnActivateFrame 325
OnActivateView 325
OnBeginPrinting 358
OnDraw 8, 326
OnEndPrinting 359
OnFilePrint 360
OnInitialUpdate 324
OnPrepareDC 8, 359
OnPreparePrinting 355
OnPrint 360
OnScroll 326
OnScrollBy 326
OnUpdate 324
CWinApp
AddDocTemplate 287, 294
CloseAllDocuments 296
CreatePrinterDC 354
GetFirstDocTemplatePosition 295
GetNextDocTemplate 295
GetPrinterDeviceDefaults 353
m_pDocManager 296
OnFileNew 296
OnFileOpen 296
OnFilePrintSetup35!
OpenDocumentFile 295
SaveAHModified 296
CWnd
Default 150
DestroyWindow 181
DrawMenuBar 383
EnableToolTips 216, 218
GetCurrent Message 149
GetStyle 123
GetWindowText 142
OnCommand 182
OnHScroll 149
OnInitMenu 389
OnInitMenuPopup 389
OnMenuSelect 390
OnNotify 182
OnVScroll 150
SetScrollPos 150
SetWindowText 119, 122, 123
D
DEFAULT_CHARSET 45, 61
DEFAULT_PALETTE 21
DEFAULT_PITCH 64
DEFAULT_QUALITY 63
DEVICE_DEFAULT_FONT 20
DKGRAY_BRUSH20,21
DL_COPYCURSOR 172
DL_MOVECURSOR 172
DL_STOPCURSOR 172
DRAFT_QUALITY 63
DRAWITEMSTRUCT 134
DSS_DEFAULT 33
DSS_DISABLED 33
DSS_MONO 33
DSS_NORMAL 32
DSS_UNION 32
DST_BITMAP 32
DST_COMPLEX 32
DST_ICON 32
DST_PREFIXTEXT 32
DST_TEXT 32
DST_INVERT 36
DT_BOTTOM 41

DT_CALCRECT 41
DT_CENTER 41
DT_EXPANDTABS 42
DT_EXTERNAL_LEADING 42
DT_LEFT 42
DT_NOCLIP 42 DT_NOPREFIX 42
DT_RIGHT 42
DT_SINGLELINE 42
DT_TABSTOP 42
DT_TOP 42
DT_VCENTER 42
DT_WORDBREAK 42, 43
E
EASTEUROPE_CHARSET 45, 61
ES_AUTOHSCROLL 116
ES_AUTOVSCROLL 116
ES_CENTER 116
ES_LEFT 116
ES_LOWERCASE 116
ES_MULTILINE117
ES_NOHIDESEL 117
ES_OEMCONVERT 117
ES_PASSWORD 117
ES_READONLY 117
ES_RIGHT 117
ES_UPPERCASE 117
ES_WANTRETURN 117
ESB_DISABLE_BOTH 154
ESB_DISABLE_LTUP 154
ESB_DISABLE_RTDN 154
ESB_ENABLE_BOTH 154
ETO_CLIPPED 40
ETO_OPAQUE 41
F
FF_DECORATIVE 45, 64
FF_DONTCARE45,64
FF_MODERN 45, 64
FF_ROMAN 45, 64
FF_SCRIPT 45, 64
FF_SWISS 45, 64
FIXED_PITCH 64
FLOODFILLBORDER 39
FLOODFILLSURFACE 39
FW_BLACK61
FW_BOLD 61
FW_DEMIBOLD 61
FW_DONTCARE 61
FW_EXTRABOLD 61
FW_EXTRALIGHT 61
FW_HEAVY 61
FW_LIGHT61
FW_MEDIUM 61
FW_NORMAL 61
FW_REGULAR 61
FW_SEMIBOLD 61
FW_THIN 61
FW_ULTRABOLD 61
FW_ULTRALIGHT 61
G
GB2312_CHARSET 45, 61
GetDialogBaseUnits 126
GetWindowLong 242
GlobalAddAtom 193
GM_ADVANCED 61
GRAY_BRUSH20,21
GREEK_CHARSET 45, 61
H
HALFTONE 23
HANGEUL_CHARSET 45, 61
HD_ITEM 234
HD_NOTIFY 236
HDF_BITMAP 234
HDF_CENTER 234
HDF_LEFT 234
HDF_OWNERDRAW 234
HDF_RIGHT 234
HDF_STRING 234
HDI_BITMAP 234
HDI_FORMAT 234
HDI_HEIGHT 234
HDI_LPARAM 234
HDI_TEXT 235
HDI_WIDTH 235
HDN_BEGINTRACK 236
HDN_DIVIDERDBLCLICK 235
HDN_ENDTRACK 236

HDN_ITEMCHANGED 235
HDN_ITEMCHANGING 235
HDN_ITEMCLICK 235
HDN_ITEMDBLCLICK 235
HDN_TRACK 236
HDS_BUTTONS 233
HDS_HIDDEN 233
HDS_HORZ 233
HDS_VERT 233
HEBREW_CHARSET 45, 61
HKCOMB_A 191
HKCOMB_C 191
HKCOMB_CA 191
HKCOMB_NONE 191
HKCOMB_S 191
HKCOMB_SA 191
HKCOMB_SC 191
HKCOMB_SCA 191
HOLLOW_BRUSH 20
HOTKEYF_ALT 190
HOTKEYF_CONTROL 190
HOTKEYF_EXT 190
HOTKEYF_SHIFT 190
HS_BDIAGONAL 58
HS_CROSS 58
HS_DIAGCROSS 58
HS_FDIAGONAL 59
HS_HORIZONTAL 59
HS_VERTICAL 59
I
ILD_BLEND25 227
ILD_BLEND50 226
ILD_NORMAL 226
ILD_OVERLAYMASK 227
ILD_TRANSPARENT 226
IMAGEINF0231
INDEXTOOVERLAYMASK 227
J
JOHAB_CHARSET 45, 61
L
LBS_DISABLENOSCROLL 126
LBS_EXTENDEDSEL 125
LBS_HASSTRINGS 125, 130, 132, 136, 170, 172
LBS_MULTICOLUMN 125
LBS_MULTIPLESEL 126, 131, 170
LBS_NOINTEGRALHEIGHT 126, 137
LBS_NOREDRAW 126
LBS_NOTIFY 124, 126
LBS_OWNERDRAWFIXED 126, 134, 170, 172
LBS_OWNERDRAWVARIABLE 126, 128, 134,137,170,172
LBS_SORT 126, 129, 130, 170
LBS_STANDARD 126
LBS_USETABSTOPS 126
LBS_WANTKEYBOARDINPUT 125, 126
LocalAlloc 121
LocalFree 121
LOGFONT 60
LTGRAY_BRUSH 20, 21
LV_COLLUMN 254
LV_HITTESTINFO 252
LV_ITEM 247
LVA_ALIGNLEFT 243
LVA_ALIGNTOP 243
LVA_DEFAULT 243
LVA_SNAPTOGRID 243
LVA_SORTASCENDING 244
LVA_SORTDESCENDING 244
LVCF_FMT 255
LVCF_SUBITEM255
LVCF_TEXT 255
LVCF_WIDTH 255
LVFI_NEARESTXY 252
LVFI_PARAM 251
LVFI_PARTIAL 251
LVFI_STRING 252
LVFI_WRAP 252
LVHT_ABOVE 253
LVHT_BELOW 253
LVHT_NOWHERE 253
LVHT_ONITEM 253

LVHT_ONITEMICON 253
LVHT_ONITEMLABEL 253
LVHT_ONITEMSTATEICON 253
LVHT_TOLEFT 253
LVHT_TORIGHT 253
LVIF_IMAGE 247
LVIF_PARAM 247
LVIF_STATE 247
LVIF_TEXT 247
LVIR_BOUNDS 250
LVIR_ICON 250
LVIR_LABEL 250
LVIS_CUT 246, 248
LVIS_DROPHILITED 246, 248
LVIS_FOOJSED 246, 248
LVIS_OVERLAYMASK 246
LVIS_SELECTED 246, 248
LVIS_STATEIMAGEMASK 246
LVN_BEGINDRAG 256, 257
LVN_BEGINLABELEDIT 256
LVN_BEGINRDRAG 256, 257
LVN_COLUMNCLICK 256
LVN_DELETEALLITEMS 256
LVN_DELETEITEM 256
LVN_ENDLABELEDIT 241, 256
LVN_GETDISPINFO 256
LVN_INSERTITEM 257
LVN_ITEMCHANGED 257
LVN_ITEMCHANGING 257
LVN_KEYDOWN 257
LVN_SETDISPINFO 257
LVNI_ABOVE251
LVNI_ALL 251
LVNI_BELOW 251
LVNI_DROPHILITED 251
LVNI_FOCUSED 251
LVNI_HIDDEN 251
LVNI_MARKED251
LVNI_SELECTED 251
LVNI_TOLEFT 251
LVNLTORIGHT251
LVS_ALIGNLEFT241
LVS_ALIGNTOP 241
LVS_AUTOARRANGE 241
LVS_EDITLABELS 241
LVS_ICON241
LVS_LIST 241
LVS_NOCOLUMNHEADER 241
LVS_NOLABELWRAP 241
LVS_NOSCROLL241
LVS_NOSORTHEADER 241
LVS_OWNERDRAWF1XED 241
LVS_REPORT241
LVS_SHAREIMAGELISTS 241
LVS_SINGLESEL 241
LVS_SMALLICON 241
LVS_SORTASCENDING 241
LVS_SORTDESCENDING 241
M
MAC_CHARSET 45, 61
MClWnd 183
MEASUREITEMSTRUCT 136
MENUITEMTEMPLATE 379
MENUITEMTEMPLATEHEADER 379
MERGECOPY 37
MERGEPAINT 37
MF_BITMAP 390
MF_BYCOMMAND 381
MF_BYPOSITION 382
MF_CHECKED 380, 390
MF_DISABLED 380, 390
MF_ENABLED 380
MF_GRAYED 380, 390
MF_MENUBARBREAK 380
MF_MENUBREAK 380
MF_MOUSESELECT 390
MF_OWNERDRAW 380, 381, 390
MF_POPUP 380, 390
MF_SEPARATOR 380, 381, 390
MF_STRING 380, 381
MF_SYSMENU 390
MF_UNCHECKED 380
MM_ANISOTROP1C 25, 26
MM_HIENGLISH 25
MM_HIMETRIC 25

MM_ISOTROPIC 25, 26
MM_LOENGLISH 25
MM_LOMETRIC 25
MM_TEXT 24, 25, 60
MM_TWIPS 25
MOD_ALT 193
MOD_CONTROL 193
MOD_SHIFT 193
N
NM_LISTVIEW 257
NM_TREEVIEW 280
NM_UPDOWNW 211
NOTSRCCOPY 37
NOTSRCERASE 37
NULL_BRUSH 20
NULL_PEN 20
O
OBM_BTNCORNERS 67
OBM_BTSIZE 67
OBM_CHECK 67
OBM_CHECKBOXES 67
OBM_CLOSE 67
OBM_COMBO 67
OBM_DNARROW 67
OBM_DNARROWD 67
OBM_DNARROWI 67
OBM_LFARROW 67
OBM_LFARROWD 67
OBM_LFARROWI 67
OBM_MNARROW 67
OBM_OLD_CLOSE 67
OBM_OLD_DNARROW 67
OBM_OLD_LFARROW 67
OBM_OLD_REDUCE 67
OBM_OLD_RESTORE 67
OBM_OLD_RGARROW 67
OBM_OLD_UPARROW 67
OBM_OLD_ZOOM 67
OBM_REDUCE 67
OBM_REDUCED 67
OBM.RESTORE 67
OBM_RESTORED 67
OBM_RGARROW 67
OBM_RGARROWD 67
OBM_RGARROWI 67
OBM_SIZE 67
OBM_UPARROW 67
OBM_UPARROWD 67
OBMJUPARROWI 67
OBM_ZOOM 67
OBM_ZOOMD 67
ODA_DRAWENTIRE 135
ODA_FOCUS 135
ODA_SELECT 135
ODS_DISABLED 135
ODS_FOCUS 135
ODS_SELECTED 135
ODT_BUTTON 135
ODT_COMBOBOX 135
ODT_LISTBOX 135
ODT_MENU 135
OEM_CHARSET45,61
OEM_FIXED_FONT 20
ON_BN_CLICKED 111
ON_BN_DOUBLECLICKED 111
ON_CBN_CLOSEUP 141
ON_CBN_DBLCLK 141
ON_CBN_DROPDOWN 141
ON_CBN_EDITCHANGE 141
ON_CBN_EDITUPDATE 142
ON_CBN_ERRSPACE 142
ON_CBN_SELCHANGE 142
ON_CBN_SELENDCANCEL 142
ON_CBN_SELENDOK 142
ON_CBN_SETFOCUS 142
ON_EN_CHANGE 115
ON_EN_ERRSPACE 115
ON_EN_HSCROLL 115
ON_EN_KILLFOCUS 116
ON_EN_MAXTEXT 116
ON_EN_SETFOCUS 116
ON_EN_UPDATE116
ON_EN_VSCROLL 116
ON_LBN_DBLCLK 124
ON_LBN_ERRSPACE 125
ON_LBN_KILLFOCUS 125

ON_LBN_SELCANCEL 124
ON_LBN_SELCHANGE 125
ON_LBN_SETFOCUS 125
ON_WM_CHARTOITEM 125
ON_WM_VKEYTOITEM 125
OPAQUE 21
OUT_CHARACTER_PRECIS 62
OUT_DEFAULT_PRECIS 62
OUT_DEVICE_PRECIS 62
OUT_OUTLINE_PRECIS 62
OUT_RASTER_PRECIS 62
OUT_STRING_PRECIS 62
OUT_STROKE_PRECIS 62
OUT_TT_ONLY_PRECIS 62
OUT_TT_PRECIS 62
OWNERREDRAW. 160
P
PATCOPY 36
PATINVERT 36
PATPAINT 36
PROOF_QUALITY 63
PS_ALTERNATE 54
PS_COSMETIC 54, 57
PS_DASH 53
PS_DASHDOT 53
PS_DASHDOTDOT 53
PS_DOT 53
PS_ENDCAP_FLAT 55
PS_ENDCAP_ROUND 55
PS_ENDCAP_SQUARE 55
PS_GEOMETRIC 54, 57
PS_INSIDEFRAME 53
PS_JOIN_BEVEL 55
PS_JOIN_MITER 55
PS_JOIN_ROUND 55
PS_NULL 53
PS_SOLID 53
PS_USERSTYLE 54, 57
R
R2_BLACK 22
R2_COPYPEN 22
R2_MASKNOTPEN 22
R2_MASKPEN 23
R2_MASKPENNOT 22
R2_MERGENOTPEN 22
R2_MERGEPEN 23
R2_MERGEPENNOT 22
R2_NOP 22
R2_NOT 22
R2_NOTCOPYPEN 22
R2_NOTMASKPEN 23
R2_ISOTMERGEPEN 23
R2_NOTXORPEN 23
R2_WHITE 22
R2_XORPEN 23
RegisterHotKey 192
RGN_AND 35, 76
RGN_COPY 35, 76
RGN_DIFF 35, 76
RGN_OR 35, 76
RGN_XOR 35, 76
RUSSIAN_CHARSET 45, 61
S
SB_BOTTOM 150,326
SB_ENDSCROLL 149, 150
SB_LEFT 149
SB_LINEDOWN 151,326
SB_LINELEFT 149
SB_LINER1GHT 150
SB_LINEUP 151,326
SB_PAGEDOWN 151,326
SB_PAGELEFT 150
SB_PAGERIGHT150
SB_PAGEUP 151,326
SB_RIGHT 150
SB_THUMBPOSITION 150, 151
SB_THUMBTRACK 150, 151, 326
SB_TOP 151,326
SBS_BOTTOMALIGN 151
SBS.HORZ 151
SBS_LEFTALIGN 152
SBS_SIZEBOX 152
SBS_SIZEBOXBOTTOMRIGHTAL IGN 152
SBS_SIZEBOXTOPLEFTALIGN 152

SBS_TOPALIGN 152
SBS_VERT 152
SC_CLOSE 388
SC_HOTKEY 388
SC_HSCROLL 388
SC_ICON 388
SC_KEYMENU 388
SC_MAXIMIZE 388
SC_MINIMIZE 388
SC_MOUSEMOVE 389
SC_MOVE 389
SC_NEXTWINDOW 389
SC_PREVWINDOW 389
SC_RESTORE 389
SC_SCREENSAVE 389
SC_SIZE 389
SC_TASKLIST 389
SC_VSCROLL 389
SC_ZOOM 388
SCROLLINFO 155, 156
SetGraphicsMode 61
SetWindowLong 242
SHIFTJIS_CHARSET 45, 61
SIF_ALL 155
SIF_DISABLENOSCROLL 155
SIF_PAGE 155
SIF_POS 156
SIF_RANGE 156
splitBar 346
splitBorder 347
splitBox 346
splitIntersection 347
SRCAND 37
SRCCOPY 37
SRCERASE 37
SRCINVERT 37
SRCPAINT 37
SS_BITMAP 102
SS_BLACKFRAME 101
SS_BLACKRECT 101
SS_CENTER 101
SS_CENTERIMAGE 102
SS.ENHMETAFILE 102
SS_GRAYFRAME 101
SS_GRAYRECT 101
SSJCON 101, 102
SS_LEFT 101
SS_LEFTNOWORDWRAP 102
SS_NOPREFIX 102
SS_RIGHT 102
SS_SIMPLE 102
SS_USERITEM 102
SS_WHITEFRAME 102
SS_WHITERECT 102
STRETCH_ANDSCANS 23
STRETCH_HALFTONE 23
STRETCH_ORSCANS 23
SYMBOL_CHARSET 45, 61
SYSTEM_FIXED_FONT 21
SYSTEM_FONT 20
T
TA_UPDATECP41,42
TB_BOTTOM 206
TB_ENDTRACK 206
TB_LINEDOWN 206
TB_LINELIP 206
TB_PAGEDOWN 206
TB..PAGEUP 206
TB_THUMBPOSITION 206
TB_THUMBTRACK 206
TB_TOP 206
TBS_AUTOTICKS 207
TBS_BOTH 207
TBS_BOTTOM 207
TBS_ENABLESELRANGE 207
TBS_HORZ 206
TBS_LEFT 207
TBS_NOTICKS 207
TBS_RIGHT 207
TBS_TOP 207
TBS_VERT 206
TEXTMETRIC 43
THAI_CHARSET45,61
TMP_CENTERALIGN 385
TMP_LEFTALIGN 385
TMP_LEFTBUTTON 385

TMP_RIGHTALIGN 385
TMP_RIGHTBUTTON 385
TMPF_DEVICE 45
TMPF_FIXED_PITCH 45
TMPF_TRUETYPE 45
TMPF_VECTOR 45
TRANSPARENT 21
TTN_NEEDTEXT216
TTS_ALWAYSTIP 220
TTS_NOPREFIX 220
TURKISH_CHARSET 45, 61
TV_HITTESTINFO 277
TV_INSERTSTRUCT 270
TV_ITEM 270
TV_SORTCB 278
TVE_COLLAPSE 275
TVE_COLLAPSERESET 275
TVE_EXPAND 275
TVE_TOGGLE 275
TVGN_CARET 275, 276
TVGN_CHILD 276
TVGN_DROPHILITE 275, 276
TVGN_FIRSTVIS1BLE 275, 276
TVGN_NEXT 276
TVGN_NEXTVISIBLE 276
TVGN_PARENT 276
TVGN_PREVIOUS 276
TVGN_PREVIOUSV1SIBLE 276
TVGN_ROOT 276
TVHT_ABOVE 277
TVHT_BELOW 278
TVHT_NOWHERE 278
TVHT_ONITEM 278
TVHT_ONITEMBUTTON 278
TVHT_ONITEMICON 278
TVHT_ONITEMINDENT 278
TVHT_ONITEMRIGHT 278
TVHT_ONITEMSTATEICON 278
TVHT
TOLEFT 278
TVHT_TORIGHT 278
TVI_FIRST 269, 270
TVI_LAST 270
TV1_SORT 269, 270
TVIF_CHILDREN271
TVIF HANDLE 271
TVIF_IMAGE271
TVIF_PARAM 271
TVIF_SELECTEDIMAGE 271
TVIF_STATE 271
TVIF_TEXT 271
TVIS_CUT 271
TVIS_DISABLED 271
TVIS_DROPHILITED 271
TVIS_EXPANDED 271
TVIS_EXPANDEDONCE 271
TVIS_FOCUSED271
TVIS_OVERLAYMASK 271
TVIS_SELECTED271
TVIS_STATEIMAGEMASK 272
TVN_BEGINDRAG 279
TVN_BEGINLABELEDIT 279
TVN_BEGINRDRAG 279
TVN_DELETEITEM 279
TVN_ENDLABELEDIT 279
TVN_GETDISPINFO 279
TVN_ITEMEXPANDED 279
TVN_ITEMEXPANDING 279
TVN_KEYDOWN 279
TVN_SELCHANGED 279
TVN_SELCHANGING 279
TVN_SETDISPINFO 279
TVS_DISABLEDRAGDROP 267
TVS_EDITLABELS 267
TVS_HASBUTTONS 267
TVS_HASLINES 267
TVS_LINESATROOT 267
TVS_SHOWSELALWAYS 267
TVSIL_NORMAL 268
TVSIL_STATE 268
U
UD_MAXVAL 214
UD_MINVAL 214
UDACCEL 215
UDN_DELTAPOS 210
UnregisterHotKey 193

V
VARIABLE_PITCH 64
VK_DOWN 206
VK_END 206
VK_HOME 206
VK_LEFT 206
VK_NEXT 206
VK_PRIOR 206
VK_RIGHT 206
VK_UP 206
W
WA_ACTIVE 325
WA_CLICKACTIVE 326
WA_INACTIVE 325
WHITE_BRUSH 20
WHITE_PEN 20
WHITENESS 36
WHITEONBLACK 23
WINDING 22
WM_HSCROLL 211
WM_KEYUP 206
WM_LBUTTONUP 206
WM_SETHOTKEY 192
WM_VSCROLL211
WS_GROUP 111
WS_TABSTOP 111

Введение

Программисты — странный народ. С одной стороны, они готовы, забыв обо всем, целыми днями возиться с новыми программными продуктами, передовыми разработками. А с другой — очень тяжело расстаются с привычными, хорошо отработанными методами и средствами программирования. Те, кто начал программировать для Windows, используя только функции API, никак не хотят переходить на использование библиотек классов, выдвигая огромное количество аргументов в пользу "своего подхода". Зачастую "медвежью услугу" в этом оказывает подбор литературы на прилавках магазинов: огромное количество (особенно в последнее время) книг, которые повторяют, с различными вариациями, один и тот же материал, и практически полное отсутствие публикаций по современным, в частности объектно-ориентированным методам программирования. Мы со своей стороны, являясь, возможно, не менее консервативными, чем другие, программистами, пытаемся продемонстрировать те преимущества, которые открываются перед программистами при использовании библиотеки классов MFC.

Библиотека классов MFC преследует две основных цели. Во-первых, скрыть от программиста то, что является скорее вспомогательным, чем обязательным аспектом при написании Windows-программ — всю "черновую работу". В частности, при использовании MFC не нужно переносить из приложения в приложение практически не изменяющуюся функцию *WinMain*, а можно непосредственно приступать к решению своей конкретной задачи.

Во-вторых, обеспечить разработчиков средством, которое, в отличие от Win32 API, более структурировано и понятно. Содержимое этого тома как раз и посвящено представлению классов и методов их использования, которое позволяет существенно упростить создание как самых простейших приложений утилитарного характера, так и сложных систем.

Как и в первом томе, здесь мы постарались "объять необъятное" и сгруппировать материал таким образом, чтобы его изложение было как можно более последовательным (как будто это возможно при описании системы, управляемой событиями!):

- Часть VII описывает практические аспекты процесса графического вывода. Знакомство с графическими возможностями Windows и библиотеки классов MFC в дальнейшем упростит понимание того, например, как следует создавать самоотображающиеся (owner-draw) элементы управления. Описываемые классы практически полностью инкапсулируют соответствующие объекты Windows, не добавляя ничего нового. Однако функции-члены этих классов группируют функции Win32 API, которые так или иначе вызываются в реализации классов. Одного этого было бы достаточно, чтобы создать и использовать классы взамен функций Win32 API. Сама реализация класса уже сводит воедино все или практически все необходимые возможности. Описывая функции-члены классов мы раскрыли все, что поддерживается в Windows. Таким образом, эту часть можно рассматривать как всеобъемлющее руководство по использованию возможностей графического вывода. Несмотря на то, что теоретические основы графического вывода подробно изложены в первом томе книги, здесь представлен большой набор примеров функций и приложений, который позволит даже новичку в программировании для Windows разобраться в том, как следует осуществлять графический вывод.

- Следующие две части (VIII и IX) посвящены классам, инкапсулирующим элементы управления Windows. В части VIII сосредоточена информация о стандартных элементах управления (Controls) и большая часть теории, касающейся всех элементов управления. Часть IX посвящена новым общим элементам управления (Common Controls). Обе эти части не только подробно излагают и описывают соответствующие классы, но содержат и информацию по самим элементам управления и описывают ряд тонких моментов, которые вряд ли можно найти в документации.

- В части X сосредоточено все, что необходимо знать о поддержке со стороны библиотеки MFC архитектуры "документ/представление" (document/view architecture). Данная архитектура напрямую не поддерживается Windows, поэтому описания только классов и функций-членов явно недостаточно. В части представлена взаимосвязь различных классов, последовательность вызовов тех или иных функций. Эти знания позволят вам в дальнейшем легко, с минимальными накладными расходами, переопределять только те функции, новое содержание которых наполнит нужным вам смыслом разрабатываемые приложения.

- Часть XI описывает классы, обеспечивающие работу с рядом ресурсов приложения, таких, например, как меню или акселераторы.

- Неотъемлемой частью тома являются приложения-примеры. Их достаточно много и они охватывают все то, что можно найти на страницах второго тома. Стремясь вместить их на одну дискету, мы воспользовались средством генерации дистрибутивов InstallShield Express VC++ 4.1 Edition. Такой подход позволяет устанавливать и переустанавливать только те примеры, которые вас заинтересуют.

Второй том книги можно считать основным с точки зрения того, что большая часть библиотеки описывается именно в первых двух томах, охватывающих 90% наиболее часто требуемых возможностей при создании Windows-приложений.

Заключительный, третий том книги предназначен, главным образом, для достаточно опытных программистов, уже знакомых с положениями, рассмотренными нами в предыдущих томах. Мы включили в него следующие темы:

- создание и работа с классами *CPropertySheet* и *CPropertyPage*, позволяющими разрабатывать блокноты свойств и мастера, а также с тесно связанным с ними классом *CTabCtrl*, инкапсулирующим общий элемент управления "вкладка";
- в завершение знакомства с общими элементами управления — работа с расширенным текстовым редактором (классы *CRichEditView* и *CRichEditCtrl*);
- многозадачность и создание многопотоковых приложений на основе классов MFC, их взаимодействие и синхронизация, а также другие, связанные с данной темой вопросы;
- механизм структурной обработки исключений и его поддержка в MFC — классы *CException* и производные от него;
- программирование трехмерной графики с использованием интересной библиотеки OpenGL;
- создание справочной системы и дистрибутива вашего приложения.

В заключение мы хотим выразить благодарность Дмитрию Шведову, написавшему главу 21. Мы выражаем благодарность редактору Татьяне Темкиной — судя по тому как сложно читать, править и исправлять даже собственный текст, ее работа не менее трудна, чем написание самой книги. Мы также благодарим всех сотрудников издательства "BHV — Санкт-Петербург", принимавших участие в подготовке рукописи к печати: Ольгу Сергиенко (компьютерная верстка), Зинаиду Дмитриеву (корректура), Николая Тверских (производство) и Елизавету Кароник (общее руководство).

Все замечания и пожелания можно присылать по адресу:

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

Часть VII

Графический интерфейс устройств (GDI)

16. Классы графического интерфейса
17. Класс CDC
18. Классы графических объектов
19. Пример приложения PQClock

Глава 16

Классы графического интерфейса

В первой и второй частях первого тома мы подробно описали механизм графического вывода. Здесь лишь напомним, что весь вывод в системе Windows сделан таким образом, чтобы унифицировать работу с такими физически различными устройствами как экран дисплея и принтеры, плоттеры и т. п. Для всех таких устройств приложение использует одни и те же функции и, в общем случае, может "не задумываться" над тем, куда будет выведена строка текста или какой-либо графический объект. Система сама распознает устройство вывода и активизирует соответствующий драйвер. Такой подход, с одной стороны, обеспечивает универсальность процесса графического вывода, а с другой — позволяет, например, создавать платы графических акселераторов, которые самостоятельно, без использования центрального процессора осуществляют преобразование команд рисования, существенно разгружая тем самым всю систему в целом.

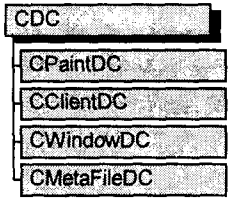
Для реализации такого подхода в Windows предусмотрен специальный объект, называемый *контекстом устройства*. Именно он хранит необходимую информацию как об устройстве вывода, так и о параметрах собственно рисования.

Win32 API имеет в своем составе большое количество функций графического вывода и настройки режимов рисования, однако их использование требует следования набору правил, в противном случае могут возникнуть ошибки времени выполнения, сбои при рисовании, потеря части содержимого окон и постепенное исчерпание системных ресурсов, что может сказаться не только на самом неправильно созданном приложении, но и на других приложениях и всей системе в целом. Последняя неприятность сплошь и рядом случалась в 16-разрядных Windows, где не освобожденные в явном виде графические ресурсы не освобождались и после завершения приложения. В 32-разрядных версиях, где более строг учет и распределение ресурсов, такое может произойти в случае продолжения использования неверно написанного приложения, а после его завершения, даже аварийного, система, за редким исключением, самостоятельно освобождает захваченные ранее ресурсы.

Реализация работы с графикой на языке C++ с использованием классов библиотеки MFC позволяет в большинстве случаев гарантировать правильное использование ресурсов (захват и освобождение) в неявном для разработчика виде. С одной стороны, библиотека обеспечивает программиста не менее мощными средствами графического вывода, но уже в виде функций-членов соответствующих классов, причем имеются и такие, у которых нет аналогов в Win32 API. С другой стороны, библиотечные классы берут на себя большую часть работы по созданию, инициализации и корректному освобождению графических ресурсов, что необходимо для правильной и эффективной реализации графического вывода. Теперь появилась возможность больше сконцентрироваться на самом процессе рисования, а не на подготовке к нему.

Перед тем как перейти к описанию тех возможностей, которые предоставляет библиотека классов MFC, скажем, что для организации графического вывода приложению необходимо работать не только с контекстами устройств, но и с рядом объектов GDI: карандашами, кистями, шрифтами и т. д. Однако функции рисования связаны именно с контекстом устройства.

Контексты устройств



В этом разделе мы проведем краткий обзор классов, инкапсулирующих различные типы контекстов устройств, поддерживаемых системой Windows. Контекст устройства — объект Windows, содержащий информацию о параметрах рисования для некоторого графического устройства, такого как экран дисплея или принтер. Физически весь графический вывод происходит, естественно, на конкретное устройство, представленное в системе контекстом, описываемым дескриптором. Последний использует в качестве параметра все функции графического вывода Win32 API, т. е. при этом задействуется объект "контекст устройства", до чего MFC-приложению нет никакого дела — оно работает с объектами классов библиотеки MFC, что существенно удобнее и надежнее.

CDC — базовый класс для всех классов, инкапсулирующих контексты устройств Windows. Объекты этого класса используются для работы со всем экраном дисплея или с таким устройством, как принтер. Кроме того, именно указатель на базовый класс передается в такие функции, как, например, *CView::OnDraw*, хотя может фактически указывать на объекты других производных классов контекстов устройств.[1]

---

[1] Параметр *CDC *pDC* функции *CView::OnDraw* на самом деле указывает на объект класса *CPaintDC*, созданный в функции *CView::OnPrepareDC*, вызываемой библиотечными функциями MFC. (В отличие от класса *CPaintDC* для объектов класса *CMetaFileDC* автоматический вызов функции *OnPrepareDC* не производится, ответственность за ее вызов лежит на разработчике приложения).

CPaintDC — объекты этого класса используются *только* в обработчике сообщения WM_PAINT, генерируемого в ответ на вызов функций *UpdateWindow* или *RedrawWindow*. Обработчиком этого сообщения обычно служит переопределяемая функция-член окна *CWnd::OnPaint*. При создании объекта класса *CPaintDC* в конструкторе вызывается функция Win32 API *BeginPaint*, а при разрушении, в деструкторе, функция *EndPaint*, также из Win32 API, обеспечивая тем самым необходимые операции для подготовки и завершения процесса графического вывода. За создание объекта класса *CPaintDC* отвечает разработчик приложения.

CClientDC — объекты этого класса обеспечивают доступ к клиентской части окна. Используется для графического вывода в любой функции, в отличие от объектов класса *CPaintDC*. При создании объекта класса *CClientDC* в конструкторе вызывается функция Win32 API *GetDC*, а при разрушении, в деструкторе, функция *ReleaseDC*, также из Win32 API, обеспечивая необходимые операции для подготовки и завершения процесса графического вывода *именно* и *только* в клиентскую часть окна. За создание объекта класса *CClientDC* отвечает разработчик приложения.

CWindowDC — объекты этого класса обеспечивают доступ ко всему окну. При создании объекта класса *CWindowDC* в конструкторе вызывается функция Win32 API *GetWindowDC*, а при разрушении, в деструкторе, функция *ReleaseDC*, также из Win32 API, обеспечивая необходимые операции для подготовки и завершения процесса графического вывода, как в *клиентскую*, так и в *неклиентскую* части окна. За создание объекта класса *CWindowDC* отвечает разработчик приложения.

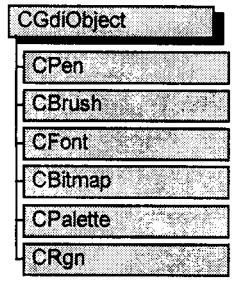
CMetaFileDC — объекты этого класса обеспечивают доступ к метафайлам[1] Windows. Вызовы функций-членов класса *CMetaFileDC* записываются в связанном с соответствующим объектом файле. Возможно создание двух типов объектов класса *CMetaFileDC*. Объекты первого типа поддерживают только функции, осуществляющие исключительно графический вывод. Такие метафайлы являются *контекстно-зависимыми*, т. к. впоследствии возможен вывод их содержимого только с учетом настроек контекста того устройства, на которое осуществляется собственно вывод. Метафайлы второго типа (*контекстно-независимые*)[2] могут содержать, в дополнение к функциям графического вывода, и функции настройки параметров рисования. Естественно, что информация из таких файлов, может быть отображена на различных устройствах одинаково.

---

[1] Метафайлы, в отличие от других графических файлов, содержат команды рисования. Вывод содержимого файла состоит в выполнении записанных команд. По умолчанию имена метафайлов имеют расширение .WMF (от Windows MetaFile).

[2] Расширенные метафайлы — Enhanced MetaFiles — имеют расширение .EMF.

Графические объекты



Настройка параметров графического вывода проводится двумя путями. Одни параметры настраиваются при помощи вызова функций Win32 API или функций-членов класса *CDC* и производных от него, другие — путем замены так называемых *графических объектов*. Библиотека MFC обеспечивает разработчиков всеми необходимыми классами, которые инкапсулируют соответствующие графические объекты Windows. Кроме того, библиотека имеет в своем составе дополнительные классы, значительно облегчающие решение ряда задач (классы *CPoint*,[1] *CSize*,[2] *CRect*,[3] *CRectTracker*[4]) или унифицирующие работу с графическими объектами как таковыми (*CGdiObject*). Вот их краткий обзор.

CGdiObject — базовый класс для всех классов, обеспечивающих интерфейс с графическими объектами Windows. Хотя этот класс и не является абстрактным, нет никакого смысла в непосредственном создании его объектов, т. к. он не ассоциирован ни с каким графическим объектом Windows.

CPen — инкапсулирует объект Windows "карандаш", который может быть назначен для контекста устройства и использоваться для определения типа и цвета линий или границ фигур.

CBrush — инкапсулирует объект Windows "кисть", который может быть назначен для контекста устройства и использоваться для определения типа и цвета заливки внутренних областей замкнутых фигур.

CFont — инкапсулирует объект Windows "шрифт", который может быть назначен для контекста устройства и использоваться при операциях вывода текстовой информации.

CBitmap — инкапсулирует объект Windows "битовый массив", который обеспечивает приложение рядом функций. Используется в том числе для работы с контекстами устройств, расположенными в памяти.[5]

CPalette — инкапсулирует объект Windows "цветовая палитра" (или просто "палитра"), который обеспечивает интерфейс между приложением и цветным устройством вывода, таким как экран дисплея. Этот интерфейс позво-

---

[1] Объекты класса *CPoint* используются для хранения пары координат (*x*, *y*).

[2] Объекты класса *CSize* используются для хранения размеров, относительного положения или других парных значений.

[3] Объекты класса *CRect* используются для хранения координат прямоугольных областей, обычно левого верхнего и правого нижнего углов.

[4] Объекты класса *CRectTracker* отображают и обеспечивают интерфейс пользователя для перемещения и изменения размеров прямоугольных объектов.

[5] См. главу 3 первого тома.

ляет приложению в полной мере использовать возможности устройства вывода, не нарушая при этом работу (графический вывод) других приложений. "Палитра" служит в Windows для определения используемых приложением цветов из набора доступных в системе.

CRgn — инкапсулирует объект Windows "регион", который хотя и относится к графическим объектам, на самом деле является скорее дополнительным объектом или, точнее, средством управления графическим выводом, определяя сложные (не только прямоугольные) области.

Как и все функции Win32 API, работающие с различными объектами Windows, функции графического вывода используют соответствующие дескрипторы. Функции классов графических объектов и контекстов устройств работают также с дескрипторами. Далее в табл. 1 показано соответствие представленных классов библиотеки MFC и типов дескрипторов объектов Windows.

Таблица 1

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

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

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

Второй способ предпочтительнее, т. к. невозможность инициализации, например, в случае исчерпания соответствующих системных ресурсов, легче определить при вызове именно функции инициализации. В противном случае, для гарантии защиты от ошибок, вернее, своевременного их обнаружения, потребуется использовать механизм обработки исключений.[1]

[1] Структурной обработке исключений и соответствующим классам библиотеки MFC посвящена отдельная часть третьего тома книги.

Для использования графического объекта для настройки соответствующих параметров рисования необходимо выполнить следующие действия:

- создать графический объект одним из двух описанных способов;
- заменить в контексте устройства вновь созданным графическим объектом текущий, сохранив при этом указатель[1] на последний;
- закончив операции рисования, восстановить графический объект в контексте устройства при помощи сохраненного указателя;
- обеспечить удаление созданного объекта по выходу из области видимости, что происходит автоматически для объектов, созданных в стеке приложения.

В случае, если один и тот же объект используется приложением достаточно часто, так что его создание и затем удаление существенно сказывается на производительности приложения в целом, возможно создание и сохранение такого объекта на все время его многократного использования. Однако перед завершением работы приложения следует обеспечить его корректное удаление, и, как следствие, освобождение ресурса, т. е. конкретного графического объекта Windows.

В следующем примере иллюстрируется методика создания и использования графических объектов при помощи классов и функций-членов библиотеки MFC.

```
void CSunView::OnDraw(CDC* pDC)
{
  // Создание с одновременной инициализацией
  CPen pen_1(PS_DOT, 1, RGB(0, 0, 0));

  try
  {
    // Создание с одновременной инициализацией
    CBrush brush(HS_CROSS, RGB(0, 0, 0));

    // Создание
    CPen pen_2;

    // Инициализация
    if (pen_2.CreatePen(PS_SOLID, 2, RGB(192, 192, 192)))
    {
      // Устанавливаем новый карандаш,
      // сохраняя указатель на текущий
      CPen* pOldPen = pDC->SelectObject(&pen_2);

      // Рисование с использованием нового карандаша
      pDC->Rectangle(...);
```

[1] Естественно, при использовании функций Win32 API приложение манипулирует не указателями, а дескрипторами, не имеющими, в частности, деструктора, и освобождение ресурса необходимо проводить в явном виде соответствующей функцией API.

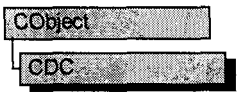
```
        // Восстанавливаем старый карандаш
        pDC->SelectObject(pOldPen);
      }
      else
      {
        // Обработка невозможности инициализации pen_2
      }
    }

    catch (CResourceException *e)
    {
      // Обработка невозможности инициализации brush
    }
}
```

Обратите внимание, что по завершении выполнения функции все созданные объекты классов *CBrush* и *CPen* будут автоматически уничтожены, и их деструкторы выполнят действия, необходимые для освобождения захваченных ресурсов. Кроме того, в примере не проводится какая-либо диагностика возможной ошибки при инициализации объекта *pen_1*, а для объекта *brush* выполняется проверка возникновения исключительной ситуации, которая возбуждается при невозможности создания объекта и имеет тип указателя на объект *CResourceException*. Причем, при возникновении исключения сразу же после выполнения конструктора управление будет передано на оператор *catch*, а промежуточные операторы будут проигнорированы.

Глава 17

Класс CDC



Класс *CDC* — базовый класс для всех классов, представляющих в библиотеке MFC контексты устройств. Класс *CDC* обеспечивает разработчика набором функций-членов для работы с устройствами, такими как экран дисплея или принтер, а также позволяющих работать с окнами Windows в целом или только с клиентской их частью. Все операции рисования выполняются только для объектов класса *CDC*. Из всех классов, обеспечивающих интерфейс с контекстами устройств, только класс *CDC* обеспечивает разработчиков всем необходимым для реализации правильного и эффективного графического вывода, остальные классы, в основном отличающиеся конструктором и деструктором, можно считать вспомогательными.

Система Windows имеет в своем составе, пожалуй, только два вида объектов, число функций для взаимодействия с которыми несоизмеримо с количеством функций для всех других объектов. Речь идет об окнах и контекстах устройств. Естественно, что библиотека классов MFC обеспечивает и соответствующие классы не менее широким набором функций-членов. Об окнах и классе *CWnd* мы подробно говорили в первом томе, сейчас же проведем обзор функциональных возможностей класса *CDC*. (В приложении 2 приводится сводка всех функций-членов класса *CDC* с краткой информацией об их назначении).

Создание объекта класса и его связь с контекстом устройства

Все классы библиотеки MFC, непосредственно связанные с объектами Windows, обеспечивают программиста более удобным и понятным интерфейсом. Часто класс имеет в своем составе конструктор по умолчанию, т. е. без параметров, который только создает объект класса, никак не связанный с объ-

ектом Windows. Так как работа с объектами Windows возможна только через дескрипторы, класс должен иметь возможность для их хранения. Теперь подробнее рассмотрим упомянутые механизмы и поддержку их в реализации класса *CDC*.

Данные класса

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

HDC CDC::**m_hDC** —
дескриптор контекста устройства графического вывода.

HDC CDC::**m_hAttribDC** —
дескриптор контекста устройства используется для получения параметров устройства.

По умолчанию *m_hDC* и *m_hAttribDC* равны. В принципе контекст устройства вывода (*m_hDC*) используется только для выполнения операций рисования, а через второй контекст запрашиваются параметры. Например, функция *SetTextColor* работает с *m_hDC*, а *GetTextColor* — с *m_hAttribDC*. Однако этот механизм имеет смысл только для объектов класса *CMetaFileDC*, когда *m_hDC* настроен на метафайл, а *m_hAttribDC* настроен на физическое устройство вывода, параметры которого используются для рисования. Этот же механизм используется и для реализации режима предварительного просмотра печати (print preview), который реализован в большом количестве приложений. Для этих целей могут быть использованы функции *SetOutputDC* и *SetAttribDC*, которые позволяют приложению связывать объект класса с различными контекстами устройств для графического вывода и получения текущих параметров.

В ряде случаев, тем не менее, может возникнуть необходимость получения параметров вывода текста от обоих контекстов. В табл. 2 показаны пары таких функций.

Таблица 2

m_hAttribDC	m_hDC
GetTextExtent	GetOutputTextExtent
GetTabbedTextExtent	GetOutputTabbedTextExtent
GetTextMetrics	GetOutputTextMetrics
GetCharWidth	GetOutputCharWidth

Конструктор

Создает объект класса, не связывая его с конкретными контекстами устройств. Для дальнейшей работы необходимо будет осуществить связывание объекта Windows, который, возможно, следует еще и создать, с объектом класса.

Инициализация

Функции этой группы либо непосредственно создают контексты для конкретных устройств и инициализируют данные класса соответствующими дескрипторами, либо только осуществляют связывание или, наоборот, разрывают связь с существующими контекстами устройств.

Из таких функций стоит выделить следующие.

Основная функция создания контекста устройства и инициализации объекта класса *CDC*:

virtual BOOL CDC::**CreateDC**(
LPCTSTR lpszDriverName,
LPCTSTR lpszDeviceName,
LPCTSTR lpszOutput,
const void* lpInitData) —

lpszDriverName — указатель на строку, содержащую имя драйвера, например, "EPSON"; lpszDeviceName — указатель на строку, содержащую имя устройства, например, "EPSON FX-1050" (этот параметр используется, если драйвер фактически поддерживает несколько устройств); lpszOutput — указатель на строку, определяющую собственно приемник — файл, символьное имя устройства или порт вывода; lpInitData — указатель на структуру *DEVMODE*, содержащую установки устройства, в случае, если значение параметра равно NULL, используются установки, заданные в системе по умолчанию.

Microsoft рекомендует имя устройства завершать символом ":", а имена устройств и портов не должны содержать ни ведущих, ни завершающих пробелов. В случае успешного завершения функция возвращает TRUE, иначе FALSE.

virtual BOOL CDC::**CreateCompatibleDC**(CDC* pDC) —

создает контекст устройства, расположенный в памяти, совместимый (подобный) заданному параметром pDC. Такие контексты позволяют выполнять операции рисования без непосредственной визуализации. Если pDC = NULL, функция создает контекст устройства, совместимый с экраном дисплея.

В момент создания контекста GDI автоматически устанавливает для него системный монохромный битовый массив размером 1 пиксель × 1 пиксель. Таким образом, операции графического вывода могут производиться только, если будет создан и установлен новый битовый массив для данного контекста устройства.

Функция может создать совместимый контекст устройства только для контекстов, поддерживающих растровые операции. Определить, поддерживает ли устройство растровые операции или нет, можно с помощью функции *GetDeviceCaps*.

Совместимые контексты используются для улучшения характеристик вывода графических объектов на экран дисплея, т. к. позволяют заранее подготовить "картинку" в памяти, а затем ее вывести полностью на экран, для чего можно воспользоваться функцией *BitBlt*, осуществляющей перенос битовых массивов между контекстами. Давайте проиллюстрируем это на примере.

```
void CMercuryFrame::OnPaint()
{
  CPaintDC dc(this);
  ...
  // создание совместимого контекста устройства
  CDC dcMem;
  dcMem.CreateCompatibleDC(&dc);
  // определение размеров клиентской области окна
  CRect rect;
  GetClientRect(&rect);
  // Создание и установка битового массива нужного размера
  // и требуемых свойств
  CBitmap bitmap, *pBitmap;
  bitmap.CreateCompatibleBitmap(&dc, rect.right, rect.bottom);
  pBitmap = dcMem.SelectObject(&bitmap);
  dcMem.Rectangle(...) // вызов функций рисования
  // копирование содержимого одного контекста в другой
  dc.BitBlt(0, 0, rect.right, rect.bottom,
    &dcMem, 0, 0, SRCCOPY);
  // восстановление первоначально созданного битового массива
  if (pBitmap)
    dcMem.SelectObject(pBitmap);
  // удаление совместимого контекста устройства
  dcMem.DeleteDC();
}
```

static CDC* PASCAL CDC::**FromHandle**(HDC hDC) —
статическая функция, возвращает указатель на объект класса CDC, связанный с задаваемым параметром hDC контекстом устройства. Обычно возвращаемое значение указывает на временный объект, т. к. функция, скорее всего, используется в нестандартной ситуации. Действительно, если приложение создано с использованием библиотеки классов MFC, то вместо дескрипторов используются все-таки объекты классов.

Так как существует вероятность того, что объект может быть временным, его нельзя использовать иначе как немедленно, т. е. в обработчике текущего сообщения. Дело в том, что временные объекты контекстов устройств удаляются автоматически объектом "приложение", созданным на базе *CWinApp*, в момент простоя или, иначе, отсутствия сообщений (idle time) посредством вызова другой статической функции *DeleteTempMap*.

Доступ к объектам рисования

Функции с префиксом *GetCurrent* возвращают указатели на установленные (текущие) графические объекты — карандаши, кисти, битовые массивы, шрифты и палитры:

CPen* CDC::**GetCurrentPen**()
CBrush* CDC::**GetCurrentBrush**()
CBitmap* CDC::**GetCurrentBitmap**()
CFont* CDC::**GetCurrentFont**()
CPalette* CDC::**GetCurrentPalette**()

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

Функции контекста устройства

Функции этой группы обеспечивают доступ к дескрипторам контекстов (*GetSafeHdc*), осуществляют сохранение и восстановление параметров контекстов устройств (*SaveDC* и *RestoreDC*). К функциям этой группы относится функция *GetDeviceCaps*, которая может быть использована для получения исчерпывающей информации о самом устройстве и соответствующем драйвере.

int CDC::**GetDeviceCaps**(int nIndex) —
параметр nIndex определяет тип требуемой информации.

Для того чтобы создать действительно универсальное приложение с точки зрения графического вывода на различные устройства, просто необходимо использовать эту функцию. Различие устройств, прежде всего, в поддержке тех или иных возможностей графического вывода, может потребовать поддержки со стороны GDI или даже самого приложения. Подробно возможные значения единственного параметра функции и возвращаемые ею значения приведены в приложении 3.

Настройка процесса рисования

Графический вывод можно разделить на две составляющие: вывод текста и вывод графических примитивов. Функции обеих групп используют (и то не всегда) в качестве параметров только координаты вывода. Все остальные

параметры, такие как цвет, стиль и т. д. задаются отдельно и действуют до их смены. Таким образом, задавать параметры вывода рисования можно для групп последовательно вызываемых функций. Альтернативой такому способу могло бы быть наличие функций со многими параметрами, что затруднило бы сам процесс программирования. К тому же, при желании всегда можно написать свою функцию, которая в зависимости от параметров, во-первых, осуществляет настройку режима рисования, и, во-вторых, производит собственно рисование. Однако в рамках Windows это вряд ли оправданно, т. к. установка ряда параметров рисования осуществляется созданием специальных объектов рисования, максимальное число которых в Windows 95 и Windows NT хоть и велико, но ограниченно. Создание таких объектов требует определенных дополнительных накладных расходов, возможность же использования текущих настроек для нескольких функций рисования, естественно, такие расходы снижает.

Класс *CDC* обеспечивает программиста следующими группами функций настройки.

Функции средств рисования

Функции этой группы определяют базовые координаты для закрашивания объектов при помощи текущей установленной кисти[1] (*GetBrushOrg* и *SetBrushOrg*) и позволяют получить информацию о системных карандашах и кистях (*EnumObjects*).

Установка объектов рисования

Функции этой группы используются для установки объектов рисования, определяющих такие параметры рисования, как цвет и стиль линий, вид закрашивания, используемый шрифт и т. д.

Основная функция установки графических объектов — *SelectObject*. Библиотека MFC в классе *CDC* определяет пять функций для различных графических объектов:

CPen* CDC::**SelectObject**(CPen* pPen)

CBrush* CDC::**SelectObject** (CBrush* pBrush)

virtual CFont* CDC::**SelectObject** (CFont* pFont)

CBitmap* CDC::**SelectObject** (CBitmap* pBitmap)

int CDC::**SelectObject** (CRgn* pRgn)

---

[1] Так как кисти представляют собой прямоугольную область 8×8 пикселей, закрашивание с их помощью областей большего размера требует определенного согласования. Иначе закраска пространства не будет ровной — например, линии штриховки будут иметь размер не больше 8 пикселей. Для осуществления такого согласования для кистей как раз и определяются базовые координаты.

Все эти функции, за исключением последней, в случае успешной установки возвращают указатель на графический объект[1] соответствующего типа, ранее установленный для контекста устройства, и NULL в противном случае. Причем указатель может указывать на временный объект и может использоваться во время обработки одного сообщения Windows. Более подробно о временных объектах рассказано при описании функции *FromHandle*.

Последняя версия функции *SelectObject* по своему действию аналогична функции *SelectClipRgn*.

Битовые массивы можно устанавливать только для контекстов устройств в памяти. Причем один битовый массив не может быть установлен одновременно больше чем для одного такого контекста. Битовый массив может быть либо монохромным, либо совместимым с контекстом устройства, т. е. созданным функцией *CBitmap::CreateCompatibleBitmap*.

Кроме созданных в приложении графических объектов можно использовать и предопределенные системные. Для таких объектов следует использовать функцию *SelectStockObject*, которая как раз и устанавливает в контексте устройства системные графические объекты следующих различных типов:

virtual CGdiObject* CDC::**SelectStockObject**(int nIndex) —
возвращает указатели на использованные ранее объекты в случае успешной замены и NULL в противном случае. Новый объект задается параметром nIndex:

BLACK_BRUSH	черная кисть
DKGRAY_BRUSH	темно-серая кисть
GRAY_BRUSH	серая кисть
LTGRAY_BRUSH	светло-серая кисть
HOLLOW_BRUSH	пустая или прозрачная кисть
NULL_BRUSH	то же, что и HOLLOW_BRUSH
WHITE_BRUSH	белая кисть
BLACK_PEN	черный карандаш
NULL_PEN	пустой или прозрачный карандаш
WHITE_PEN	белый карандаш
ANSI_FIXED_FONT	непропорциональный системный шрифт ANSI
ANSI_VAR_FONT	пропорциональный системный шрифт ANSI
DEVICE_DEFAULT_FONT	шрифт, зависимый от устройства
OEM_FIXED_FONT	непропорциональный шрифт OEM
SYSTEM_FONT	системный шрифт, используемый по умолчанию для отображения пунктов меню, текста элементов управления и т. д.; в версиях Windows, начиная с 3.0, системный шрифт пропорциональный

[1] Правильнее будет называть его объектом класса графического объекта.

SYSTEM_FIXED_FONT	непропорциональный системный шрифт, используется для совместимости с предыдущими версиями Windows
DEFAULT_PALETTE	основная цветовая палитра, содержащая 20 цветов

Системные кисти DKGRAY_BRUSH, GRAY_BRUSH и LTGRAY_BRUSH могут использоваться только для окон, имеющих стили CS_HREDRAW и CS_VREDRAW. Связано это с тем, что базовые координаты системных кистей[1] не могут настраиваться, и при изменении размеров или положения окон их содержимое может быть испорчено.

Функции настройки цветов и палитр

Функции этой группы позволяют установить новую цветовую палитру (*SelectPalette*), перерисовать окно, заменяя ранее установленные цвета (*UpdateColors*), а также создать полутоновую кисть (*GetHalftoneBrush*), цвет которой получается в результате смешивания текущих цвета фона (background) и цвета, используемого для вывода текстовых строк (foreground).

Режимы рисования

Функции этой группы используются для настройки различных режимов рисования и закрашивания замкнутых областей. Основные функции мы рассмотрим подробнее, т. к. их значение для реализации графического вывода чрезвычайно важно. Все функции реализованы парами: с префиксом *Get* — позволяет получить текущее значение параметра, с префиксом *Set* — устанавливает новое значение и возвращает предыдущее. Далее следует описание наиболее часто используемых функций.

int CDC::**SetBkMode**(int nBkMode) —
используется для установки режима закрашивания фона. Возможные режимы задаются параметром nBkMode, который может принимать следующие значения:

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

int CDC::**SetPolyFillMode**(int nPolyFillMode) —
используется для установки режима закрашивания многоугольников. Возможные режимы задаются параметром nPolyFillMode:

[1] См. примечание на странице 19.

ALTERNATE	режим закрашивания внутренних областей, находящихся между первой и второй, третьей и четвертой и т. д. линиями контура; задан по умолчанию
WINDING	режим, при котором закрашиваются все внутренние области

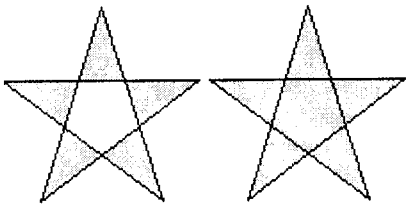


Рис. 1. Отличие режимов закрашивания: ALTERNATE — слева, WINDING — справа

Для установки режима рисования или заливки используется функция *SetROP2*. Название функции расшифровывается как "установка режима растровых (пиксельных) операций, использующих в качестве параметров два пикселя". Такими пикселями являются соответствующие друг другу точки экрана и отображаемого объекта.

int CDC::**SetROP2**(int nDrawMode)

Возможные режимы задаются параметром nDrawMode, который может принимать следующие значения:

R2_BLACK	пиксели будут черного цвета
R2_WHITE	пиксели будут белого цвета
R2_NOP	пиксели останутся без изменений
R2_NOT	пиксели будут иметь инверсный по отношению к экрану цвет
R2_COPYPEN	пиксели будут иметь цвет, заданный для текущего карандаша
R2_NOTCOPYPEN	пиксели будут иметь инверсный цвет по отношению к цвету текущего карандаша
R2_MERGEPENNOT	цвет пикселей получается в результате комбинирования цвета карандаша и инверсии цвета экрана ("цвет пикселя" = (NOT "цвет пикселя экрана") OR "цвет карандаша")
R2_MASKPENNOT	цвет пикселей получается в результате комбинирования цвета карандаша и инверсии цвета экрана ("цвет пикселя" = (NOT "цвет пикселя экрана") AND "цвет карандаша")
R2_MERGENOTPEN	цвет пикселей получается в результате комбинирования цвета экрана и инверсии цвета карандаша ("цвет пикселя" = (NOT "цвет карандаша") OR "цвет пикселя экрана")
R2_MASKNOTPEN	цвет пикселей получается в результате комбинирования цвета экрана и инверсии цвета карандаша ("цвет пикселя" = (NOT "цвет карандаша") AND "цвет пикселя экрана")

R2_MERGEPEN цвет пикселей получается в результате комбинирования цвета экрана и цвета карандаша ("цвет пикселя" = "цвет карандаша" OR "цвет пикселя экрана")

R2_NOTMERGEPEN цвет пикселей получается в результате инверсии R2_MERGEPEN цвета ("цвет пикселя" = NOT("цвет карандаша" OR "цвет пикселя экрана"))

R2_MASKPEN цвет пикселей получается в результате комбинирования цвета экрана и цвета карандаша ("цвет пикселя" = "цвет карандаша" AND "цвет пикселя экрана")

R2_NOTMASKPEN цвет пикселей получается в результате инверсии R2_MASKPEN цвета ("цвет пикселя" = NOT("цвет карандаша" AND "цвет пикселя экрана"))

R2_XORPEN цвет пикселей получается в результате комбинирования цвета экрана и цвета карандаша ("цвет пикселя" = "цвет карандаша" XOR "цвет пикселя экрана")

R2_NOTXORPEN цвет пикселей получается в результате инверсии R2_XORPEN цвета ("цвет пикселя" = NOT("цвет карандаша" XOR "цвет пикселя экрана"))

Вышеперечисленные режимы рисования линий поддерживаются только растровыми устройствами.

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

int CDC::**SetStretchBltMode**(int nStretchMode) —

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

STRETCH_ANDSCANS или **BLACKONWHITE** цвет пикселей получается в результате комбинирования цвета удаляемых пикселей и остающихся с использованием операции AND; для монохромных изображений черные пиксели имеют приоритет над белыми

STRETCH_DELETESCANS или **COLORONCOLOR** режим простого удаления пикселей без проведения попыток сохранить информацию об их цвете

STRETCH_HALFTONE или **HALFTONE** при сжатии цвет пикселей результирующего блока "в среднем" сохраняется

STRETCH_ORSCANS или **WHITEONBLACK** цвет пикселей получается в результате комбинирования цвета удаляемых пикселей и остающихся с использованием операции OR; для монохромных изображений белые пиксели имеют приоритет над черными

К функциям этой же группы относятся функции для назначения и запроса цветов. Одни используются для цвета символов текста (функции *SetTextColor* и *GetTextColor*), а другие — для цвета фона (функции *SetBkColor* и *GetBkColor*).

Цвет для символов текста используется только в случае, если текстовая строка выводится при помощи функций *CDC::TextOut* или *CDC::ExtTextOut*.

Цвет фона используется для заливки промежутков несплошных линий, областей, закрашенных штрихованными кистями, а также в просветах между символами текста, однако для линий, выведенных с использованием карандашей, созданных с флагами PS_GEOMETRIC или PS_COSMETIC, цвет фона не используется.

Режим отображения

До сих пор, говоря о задании координат, мы подразумевали, что речь идет о смещении относительно левого верхнего угла экрана (screen coordinates) или левого верхнего угла клиентской области окна (client coordinates). В принципе, для всего, что касается окон в целом и оконных сообщений, например WM_MOVE, WM_MOUSEMOVE, это правильно. Однако всегда ли удобно рисовать линию "по точкам", когда, во-первых, все перевернуто с ног на голову — координаты по вертикали растут вниз, во-вторых, приходится постоянно заботиться о масштабе, и, в-третьих, приходится помнить формулы вычисления координат точек при повороте и смещении, учитывая недостатки такой системы координат?

Система Windows обеспечивает программиста возможностью создания своей системы координат. Для этого можно, манипулируя различными параметрами, задавать направление и масштаб осей, а также начало координат.

Настройка режимов отображения

Функции этой группы устанавливают и настраивают систему координат, которая используется всеми функциями вывода — координаты вывода задаются именно в логических единицах. Кроме единиц по осям *x* и *y*, функции этой группы задают направление осей и начало координат. Практически все функции группы реализованы парами. Функции с префиксом *Get* позволяют получить текущее значение параметра, а с префиксом *Set* устанавливают новое значение и возвращают предыдущее. По умолчанию начало координат находится в левом верхнем углу области вывода, (прежде всего, окна), ось *x* направлена слева направо, а ось *y* — сверху вниз. Единица измерения — один пиксель (режим MM_TEXT). Кстати сказать, именно поэтому на различных устройствах, вследствие различий в размерах пикселя, фактический (визуальный) размер по оси *x* может не соответствовать размеру по оси *y*.

Для настройки системы координат Windows использует два понятия — *физическая область вывода* (viewport), координаты и размеры которой задаются в

физических единицах — пикселях, и *логическая область вывода* (window), координаты и размеры которой задаются в логических единицах, определяемых режимом рисования.

Теперь давайте познакомимся с режимами отображения, которые поддерживает Windows. Всего их восемь, и основная функция, которая позволяет выбрать режим, а значит и систему координат, — *SetMapMode*:

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

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

MM_TEXT [1]	одна логическая единица равна одному пикселю, ось *x* направлена вправо, ось *y* направлена вниз; режим задан по умолчанию
MM_HIENGLISH	одна логическая единица равна 0.001 дюйма, ось *x* направлена вправо, ось *y* направлена вверх
MM_HIMETRIC	одна логическая единица равна 0.01 миллиметра, ось *x* направлена вправо, ось *y* направлена вверх
MM_LOENGLISH	одна логическая единица равна 0.01 дюйма, ось *x* направлена вправо, ось *y* направлена вверх
MM_LOMETRIC	одна логическая единица равна 0.1 миллиметра, ось *x* направлена вправо, ось *y* направлена вверх
MM_TWIPS	одна логическая единица — твипс (twips) — равна 1/20 точки (point) или 1/1440 дюйма,[2] ось *x* направлена вправо, ось *y* направлена вверх
MM_ANISOTROPIC	режим позволяет настраивать, с помощью функций SetWindowExt и SetViewportExt, размерность (отдельно для каждой из осей), их направления и начало отсчета
MM_ISOTROPIC	режим позволяет настраивать, с помощью функций SetWindowExt и SetViewportExt, размерность осей, их направления и начало отсчета, однако единица оси *x* равна единице оси *y*, таким образом, при необходимости GDI настраивает размерность по осям, обеспечивая соотношение 1:1

Отметим, что только режимы MM_HIENGLISH, MM_HIMETRIC, MM_LOENGLISH, MM_LOMETRIC и MM_TWIPS обеспечивают прило-

---

[1] Режим назван таким образом не потому, что он наиболее удобен для вывода текста, а потому, что направление осей — суть его система координат — соответствует способу чтения, принятому в европейских языках — слева направо и сверху вниз.

[2] Исторически в полиграфии используются такие единицы измерения как *точки* (points) и *твипсы* (twips), размеры которых соответственно равны (примерно) 1/72 и 1/1440 дюйма, мы же в результате вынуждены использовать понятие пиксель, говоря о точке экрана. К сожалению, о богатстве русского языка в специальных областях говорить не приходится.

жения физически понятными единицами измерения с одной стороны, а с другой — определенной независимостью от устройств вывода.

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

И физическая и логическая области вывода характеризуются точкой, определяющей начало координат. Для настройки начала координат физической области вывода используются функции:

```
virtual CPoint CDC::SetViewportOrg(
   int x,
   int y)
```

или

```
virtual CPoint CDC::SetViewportOrg(POINT point)
```

Для настройки начала координат логической области вывода используются следующие функции:

```
CPoint CDC::SetWindowOrg(
   int x,
   int y)
```

или

```
CPoint CDC::SetWindowOrg(POINT point)
```

Все четыре функции в качестве параметров получают новые значения для установки начала координат, а возвращают предыдущие значения. Причем если для физической области вывода указываются координаты, которые в дальнейшем будут являться началом координат, то для логической области вывода указываются *логические* координаты, которые в дальнейшем будут соответствовать левому верхнему углу логической области вывода.

Из вышесказанного следует, что настраивать начало координатной сетки есть смысл только для одной из областей, т. к. система координат логической области вывода относительна системе координат физической области. Поясним сказанное на примерах.

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

```
void CVenusFrame::OnPaint()
{
  CPaintDC dc(this);

  CRect rect;
  GetClientRect(rect);
```

```
  dc.SetViewportOrg(rect.Width() / 2, rect.Height() / 2);

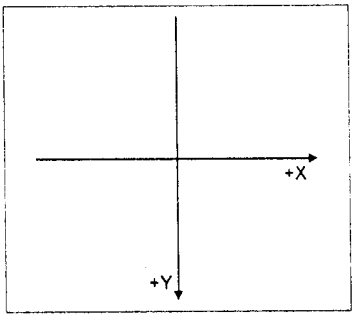
  ...
}

void CEarthFrame::OnPaint()
{
  CPaintDC dc(this);

  CRect rect;
  GetClientRect(rect);

  dc.DPtoLP(rect);
  dc.SetWindowOrg(-rect.Width() / 2, -rect.Height() / 2);

  ...
}
```



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

```
void CMarsFrame::OnPaint()
{
  CPaintDC dc(this);

  CRect rect;
  GetClientRect(rect);

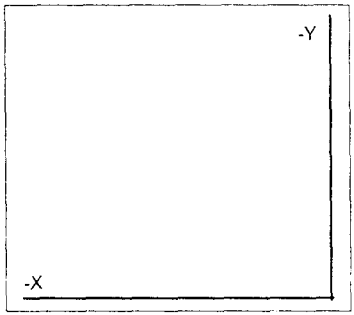
  dc.SetViewportOrg(rect.Width() / 2, rect.Height() / 2);

  //
  // вызовы следующих двух функций необходимы в случае,
  // если текущий режим отличен от MM_TEXT (см. далее),
  // т. к. функция SetWindowOrg работает с логическими
  // координатами, а в результате преобразования, из-за
  // несоответствия направления осей, длина и ширина области
  // может быть вычислена неверно
  //
```

```
  dc.DPtoLP(rect);
  rect.NormalizeRect();

  dc.SetWindowOrg(-rect.Width() / 2, -rect.Height() / 2);

  ...
}
```



Кроме начала координат каждая из областей вывода характеризуется еще, что вполне естественно, размерами (extents), которые используются *только* для определения коэффициента масштабирования.[1] Коэффициент для каждой из осей находится как отношение соответствующих размеров областей вывода. Однако для большинства режимов коэффициент масштабирования изменить невозможно.

Для настройки размеров физической области вывода используются функции:

```
virtual CSize CDC::SetViewportExt(
   int cx,
   int cy)
```

или

```
virtual CSize CDC::SetViewportExt(SIZE size)
```

Для настройки размеров логической области вывода используются функции:

```
virtual CSize CDC::SetWindowExt(
   int cx,
   int cy)
```

или

```
virtual CSize CDC::SetWindowExt(SIZE size)
```

Все четыре функции в качестве параметров получают новые значения-размеры по осям. Эти значения используются для преобразования логических координат в физические. Кроме того, настройка размеров для физиче-

[1] Размеры области, куда может осуществляться вывод, определяются размерами клиентской части окна.

ского окна вывода позволяет управлять направлением осей. Поскольку после выбора одного из режимов ось *x* направлена слева направо, а ось *y* — сверху вниз, то для изменения направления, например, оси *y*, размер должен быть задан с отрицательным значением. Все функции возвращают предыдущие значения, причем они могут отличаться от тех, что были фактически заданны соответствующими функциями, т. к. Windows настраивает их значения в зависимости от конкретных возможностей или типа устройства.

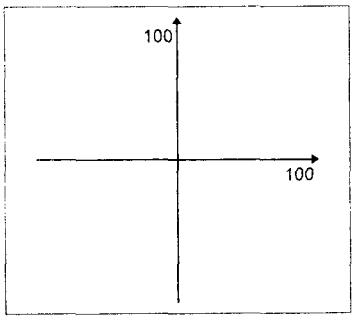
Далее приводится код, который помещает начало координат в середину клиентской области окна, ось *y* направлена вверх, а минимальный из размеров по горизонтали или вертикали равен 200 логическим единицам. Причем вывод может осуществляться и по координатам, большим 100 или меньшим −100 соответственно.[1]

```
void CSaturnFrame::OnPaint()
{
  CPaintDC dc(this);

  CRect rect;
  GetClientRect(rect);

  dc.SetMapMode(MM_ISOTROPIC);
  dc.SetWindowExt(100, 100);
  dc.SetViewportExt(rect.Width(), -rect.Height());
  dc.SetViewportOrg(rect.Width() / 2, rect.Height() / 2);

  ...
}
```



Кроме вышеперечисленных функций установки начала координат и размеров осей класс *CDC* содержит ряд функций для модификации текущих установок (см. приложение 2).

---

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

Преобразование координат

В описании большинства функций рисования указано, что координаты задаются в логических единицах, величина которых зависит от установленного режима. Однако ряд функций продолжают понимать только физические единицы устройства — пиксели. Для таких функций необходимо осуществлять преобразование координат.

Функции этой группы предназначены для преобразования логических координат в физические и наоборот (*LPtoDP* и *DPtoLP*). Кроме того, четыре функции *DPtoHIMETRIC*, *LPtoHIMETRIC*, *HIMETRICtoDP* и *HIMETRICtoLP* используются для преобразования координат при передаче размеров объекта OLE и при их получении.

Функции рисования

Итак, параметры заданы, объекты рисования выбраны, режимы установлены — можно рисовать. Давайте рассмотрим, какие возможности предоставляет нам класс *CDC* для рисования различных графических объектов.

Отображение регионов

Функции этой группы позволяют рисовать рамки (*FrameRgn*), закрашивать (*FillRgn* и *PaintRgn*) и инвертировать цвета регионов (*InvertRgn*).

Отсечения

Функции этой группы добавляют/исключают области отсечения, которые используются для ограничения области перерисовки (*SelectClipRgn*, *ExcludeClipRect*, *ExcludeUpdateRgn* и *IntersectClipRect*). Кроме того, использование части функций группы, например, *GetClipBox* и *PtVisible*, позволяет определять текущую область отсечения для программного исключения перерисовки части рисунка, что в ряде случаев существенно сокращает время вывода. Для повторного вывода сложного объекта могут потребоваться не только многократные вызовы функций настройки различных параметров и функций собственно вывода, но и длительный по времени процесс вычисления, например, для вывода контуров трехмерных объектов.

Рисование линий

Функции этой группы предназначены для рисования прямых (*LineTo*), гладких (*Arc*, *ArcTo* и *AngleArc*) и ломаных линий (*Polyline*, *PolylineTo*, *PolyPolyline*, *PolyBezier*, *PolyBezierTo* и *PolyDraw*) установленным в момент рисования карандашом. Часть функций меняют при этом текущее положение карандаша (*LineTo*, *ArcTo*, *PolylineTo* и *PolyBezierTo*), другие — нет. Даже если в резуль-

тате рисования контур получается замкнутым, закрашивание не осуществляется.[1]

Базовые функции рисования

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

Остановимся на функции, вернее, на функциях, *DrawState* подробнее. Сразу надо сказать, что поддержка этих функций есть только в Windows 95 и Windows NT 4.0. Вот определения всех версий функции.

Отображение битовых массивов

```
BOOL CDC::DrawState(
   CPoint pt,
   CSize size,
   HBITMAP hBitmap,
   UINT nFlags,
   HBRUSH hBrush = NULL)
```

```
BOOL CDC::DrawState(
   CPoint pt,
   CSize size,
   CBitmap* pBitmap,
   UINT nFlags,
   CBrush* pBrush = NULL)
```

Отображение пиктограмм

```
BOOL CDC::DrawState(
   CPoint pt,
   CSize size,
   HICON hIcon,
   UINT nFlags,
   HBRUSH hBrush = NULL)
```

```
BOOL CDC::DrawState(
   CPoint pt,
   CSize size,
   HICON hIcon,
   UINT nFlags,
   CBrush* pBrush = NULL)
```

Отображение строк

```
BOOL CDC::DrawState(
   CPoint pt,
   CSize size,
   LPCTSTR lpszText,
   UINT nFlags,
   BOOL bPrefixText = TRUE,
   int nTextLen = 0,
   HBRUSH hBrush = NULL)
```

```
BOOL CDC::DrawState(
   CPoint pt,
   CSize size,
   LPCTSTR lpszText,
   UINT nFlags,
   BOOL bPrefixText = TRUE,
   int nTextLen = 0,
   CBrush* pBrush = NULL)
```

---

[1] Следующие функции на сегодняшний день поддерживаются только в рамках Windows NT: *ArcTo*, *AngleArc*, *PolyDraw*.

"Управляемый вывод" изображений

```
BOOL CDC::DrawState(
  CPoint pt,
  CSize size,
  DRAWSTATEPROC lpDrawProc,
  LPARAM lData,
  UINT nFlags,
  HBRUSH hBrush = NULL)
```

```
BOOL CDC::DrawState(
  CPoint pt,
  CSize size,
  DRAWSTATEPROC lpDrawProc,
  LPARAM lData,
  UINT nFlags,
  CBrush* pBrush = NULL)
```

Общие параметры функций

Общие два первых параметра определяют точку привязки (левый верхний угол прямоугольной области вывода — параметр *pt*) и размеры области вывода (параметр size).

Параметр *nFlags* определяет тип изображения и может принимать одно из следующих значений:

DST_BITMAP — битовый массив; в последних двух версиях функции младшее слово параметра lData содержит дескриптор битового массива

DST_COMPLEX — пользовательский тип изображения; для вывода изображения используется функция обратного вызова (callback), заданная параметром lpOutputFunc

DST_ICON — пиктограмма; в последних двух версиях функции младшее слово параметра lData содержит дескриптор пиктограммы

DST_PREFIXTEXT — символьная строка; если текст содержит символ "&", при выводе следующий за ним символ выводится с подчеркиванием, символ "&" не выводится; в последних двух версиях функции параметр lData содержит адрес символьной строки, а параметр wData содержит ее длину (количество символов), если этот параметр равен 0, то для определения длины строки она должна завершаться нуль-символом

DST_TEXT — символьная строка; в последних двух версиях параметр lData содержит адрес символьной строки, а параметр wData содержит ее длину (количество символов), если этот параметр равен 0, то для определения длины строки она должна завершаться нуль-символом

Из названия функции не совсем ясно, причем тут состояние (state). Параметр *nFlags* может комбинировать со значением типа изображения следующие флаги состояния, задающие вид модификации изображения:

DSS_NORMAL — модификации отсутствуют

DSS_UNION — объединение части изображения с фоном, т. е. частичное разрушение (dithering) изображения (например, строки выводятся серым цветом)

DSS_DISABLED	выдавливание (embossing) изображения — эффект рельефной каменной плиты
DSS_DEFAULT[1]	вывод изображения в более отчетливой (контрастной) форме
DSS_MONO	область вывода заполняется при помощи заданной кисти (параметры hBrush или pBrush)

Для всех состояний, кроме DSS_NORMAL, изображение конвертируется в монохромное перед тем как к нему применяются эффекты.

Отображение эллипсов и многоугольников

Функции этой группы применяются для рисования сложных фигур, таких как эллипсы (*Ellipse*), сегменты (*Chord*), сектора (*Pie*), многоугольники (*Rectangle*, *RoundRect*, *Polygon* и *PolyPolygon*) и т. д. Функции используют для рисования текущие карандаш и кисть. В случае необходимости выделения прямоугольной области можно использовать функцию *DrawFocusRect*, которая выполняет над содержимым области операцию XOR.

Контуры

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

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

Только часть функций рисования используется для создания контура. Вот они:

AngleArc	LineTo	Polyline
Arc	MoveTo	PolylineTo
ArcTo	Pie	PolyPolygon
Chord	PolyBezier	PolyPolyline
CloseFigure	PolyBezierTo	Rectangle
Ellipse	PolyDraw	RoundRect
ExtTextOut	Polygon	TextOut

[1] В данном случае это состояние названо так не с точки зрения программирования: по умолчанию используется флаг DSS_NORMAL, значение которого равно нулю и, следовательно, может в явном виде не задаваться.

К сожалению, не все эти функции влияют на содержимое контура так, как бы хотелось — кроме функций вывода текста это только функции рисования линий, ломаных и кривых Безье. Связано это с тем, что контур хранит только координаты вершин составляющих его линий. Например, дуга при прорисовке контура будет заменена прямой линией. Кривые Безье, по сути, являются совокупностями небольших прямых линий или, иначе, ломаными.

Приведем пример создания и использования контуров для закраски символов текста, что может быть чрезвычайно полезно, т. к. существенно упрощает сложные операции, реализованные только в мощных графических редакторах.

```
void CClipPathFrame::OnPaint()
{
  CPaintDC dc(this);

  CFont font;
  font.CreateFont(80, 30, 0, 0, FW_BOLD, 0, 0, 0, DEFAULT_CHARSET,
    OUT_DEFAULT_PRECIS, CLIP_DEFAULT_PRECIS, PROOF_QUALITY,
    VARIABLE_PITCH | FF_DONTCARE, "Arial");

  // назначаем текущим ранее подготовленный шрифт
  CFont *pFont = dc.SelectObject(&font);

  // т. к. строка нам понадобится несколько раз,
  // создадим объект типа CString, что позволит,
  // кроме всего прочего, использовать версии
  // функций с меньшим количеством параметров
  CString csText = _T("SelectClipPath");

  // узнаем размеры строки, которая будет
  // выведена с использованием текущего шрифта
  CSize sz = dc.GetTextExtent(csText);

  // сообщаем системе о создании контура
  dc.BeginPath();

  // строка не будет выведена на экран -
  // на основе ее очертания будет создан контур
  dc.TextOut(10, 10, csText);

  // завершаем создание контура
  dc.EndPath();

  // на основе контура создаем и устанавливаем
  // область отсечения
  dc.SelectClipPath(RGN_XOR);

  // выводим горизонтальные линии с изменяемым
  // расстоянием между ними
  double delta = 2.0;
```

```
  for (int i = 0; i < sz.cy; i += (int)delta)
  {
    dc.MoveTo(10, 10+i);
    dc.LineTo(10+sz.cx, 10+i);

    delta *= 1.025;
  }

  // устанавливаем ранее использованный шрифт
  dc.SelectObject(pFont);

  // сообщаем системе о том, что далее областью
  // отсечения будет вся клиентская область окна
  dc.SelectClipRgn(NULL);
}
```

В результате выполнения этого кода мы получим строку, заливка символов которой не может быть получена обычными средствами.

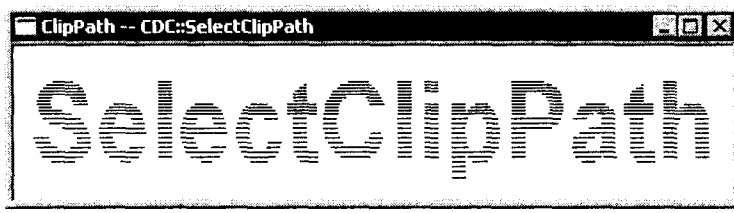


Рис. 2. Результат работы функции CClipPathFrame::OnPaint

Немного подробнее опишем функцию *CDC::SelectClipPath*, использованную в примере:

BOOL CDC::**SelectClipPath**(int nMode)

Параметр nMode задает один из пяти режимов создания новой области отсечения на основе текущей области и текущего контура. Далее перечислены возможные режимы (они также используются и в функции создания новой области отсечения с использованием регионов CDC::SelectClipRgn):

RGN_AND	новая область отсечения включает пересечение текущей области и текущего контура
RGN_COPY	новая область отсечения совпадает с текущим контуром
RGN_DIFF	новая область отсечения состоит из текущей, исключая пересечение с текущим контуром
RGN_OR	новая область отсечения является объединением текущей области и текущего контура
RGN_XOR	новая область отсечения является объединением текущей области и текущего контура, исключая область пересечения

Отображение битовых массивов

Если можно было бы рисовать только математически заданные фигуры, графические возможности Windows были бы существенно ограниченны (конечно, реализации отображения и копирования только растровых (точечных) изображений тоже недостаточно). Система Windows, и вместе с ней класс *CDC*, предоставляют набор достаточно мощных функций, которые позволяют выводить рисунки, дополнительно масштабируя их, или осуществляя их вращение. К сожалению, последняя возможность реализована только в рамках Windows NT (функция *PlgBlt*).

Функции, использующие битовые массивы

Функции этой группы используются для отображения отдельных точек (*GetPixel*, *SetPixel* и *SetPixelV*) или массивов точек, задаваемых битовыми массивами. Отдельно отметим функции копирования и преобразования битовых массивов, а также функции закрашивания замкнутых областей.

Функция *PatBlt* используется для создания массива точек, который формируется в результате комбинации текущей кисти и содержимого области вывода (текущего заполнения):

```
BOOL CDC::PatBlt(
  int x,
  int y,
  int nWidth,
  int nHeight,
  DWORD dwRop)
```

Параметр dwRop определяет режим комбинирования текущей кисти и содержимого области, задаваемой параметрами x, y, nWidth и nHeight. Возможны следующие режимы:

PATCOPY	заполнение области при помощи текущей кисти
PATINVERT	комбинирование текущего заполнения с кистью при помощи оператора XOR
DSTINVERT	инвертирование текущего заполнения
BLACKNESS	заданная область закрашивается сплошным черным цветом
WHITENESS	заданная область закрашивается сплошным белым цветом
PATPAINT	комбинирование инвертированного битового массива с текущей кистью при помощи оператора OR и комбинирование результата этой операции с текущим заполнением при помощи оператора OR

Не все устройства поддерживают выполнение функции *PatBlt*. Для определения поддержки необходимо проверить результат вызова функции *GetDeviceCaps* с параметром RASTERCAPS (флаг RC_BITBLT).

Функция *BitBlt* копирует битовый массив одного контекста устройства в другой:

```
BOOL CDC::BitBlt(
   int x,
   int y,
   int nWidth,
   int nHeight,
   CDC* pSrcDC,
   int xSrc,
   int ySrc,
   DWORD dwRop)
```

Параметр dwRop определяет растровую операцию, используемую при копировании битового массива из контекста устройства, заданного параметром pSrcDC в область, задаваемую параметрами x, y, nWidth и nHeight. Параметры xSrc и ySrc задают верхний левый угол области контекста устройства источника, который и определяет копируемый битовый массив. Для операций, не требующих битового массива в качестве источника, параметр pSrcDC должен быть равен NULL.

Возможно использование следующих растровых операций:

BLACKNESS	то же, что и для функции PatBlt
DSTINVERT	то же, что и для функции PatBlt
MERGECOPY	комбинирование текущего заполнения с текущей кистью при помощи оператора AND
MERGEPAINT	комбинирование инвертированного битового массива с текущим заполнением при помощи оператора OR
NOTSRCCOPY	копирование инвертированного битового массива
NOTSRCERASE	инвертирование результата комбинирования текущего заполнения с битовым массивом при помощи оператора OR
PATCOPY	то же, что и для функции PatBlt
PATINVERT	то же, что и для функции PatBlt
PATPAINT	то же, что и для функции PatBlt
SRCAND	комбинирование текущего заполнения и битового массива при помощи операции AND
SRCCOPY	заполнение заданной области битовым массивом
SRCERASE	комбинирование битового массива и инвертированного текущего заполнения при помощи операции AND
SRCINVERT	комбинирование битового массива и текущего заполнения при помощи операции XOR
SRCPAINT	комбинирование битового массива и текущего заполнения при помощи операции OR
WHITENESS	то же, что и для функции PatBlt

Данная функция существенно быстрее выполняется для окон, имеющих стили CS_BYTEALIGNWINDOW или CS_BYTEALIGNCLIENT.[1] Если размеры принимающей области больше битового массива источника, GDI вызывает функцию *StretchBlt*, т. е. происходит масштабирование.

Не все устройства поддерживают выполнение функции *BitBlt*. Для определения поддержки необходимо проверить результат вызова функции *GetDeviceCaps* с параметром RASTERCAPS (флаг RC_BITBLT).

Функция *StretchBlt* копирует битовый массив одного контекста в другой, изменяя размеры.

```
BOOL CDC::StretchBlt(
  int x,
  int y,
  int nWidth,
  int nHeight,
  CDC* pSrcDC,
  int xSrc,
  int ySrc,
  int nSrcWidth,
  int nSrcHeight,
  DWORD dwRop)
```

Функция копирует битовый массив из контекста устройства, заданного параметром pSrcDC, в область, задаваемую параметрами x, y, nWidth и nHeight. Параметры xSrc и ySrc определяют верхний левый угол области контекста устройства источника, а параметры nSrcWidth и nSrcHeight — размеры битового массива. Если параметры nWidth и nSrcWidth имеют разные знаки, то исходный битовый массив отображается зеркально по оси x. Если параметры nHeight и nSrcHeight имеют разные знаки, то исходный битовый массив отображается зеркально по оси y.

При уменьшении или увеличении образа используется режим преобразования заданный функцией CDC::StretchBltMode.

Параметр dwRop принимает такие же значения, как и для функции BitBlt, хотя операции, не использующие битовые массивы, для этой функции не имеют смысла — масштабировать будет нечего.

Функция осуществляет масштабирование битового массива в памяти и только потом выполняет копирование (с учетом заданного растрового режима) в контекст устройства.

Не все устройства поддерживают выполнение функции *StretchBlt*. Для определения поддержки необходимо проверить результат вызова функции *GetDeviceCaps* с параметром RASTERCAPS (флаг RC_STRETCHBLT).

Следующая функция обеспечивает закрашивание произвольных областей с использованием текущей кисти:

---

[1] Для указания этих стилей необходимо зарегистрировать класс окна при помощи глобальной функции MFC *::AfxRegisterWndClass* (См. Том I).

BOOL CDC::**ExtFloodFill**(
int x,
int y,
COLORREF crColor,
UINT nFillType) —

возвращает TRUE при успешном завершении и FALSE в случае возникновения ошибки. Закрашивание невозможно, если пиксель, координаты которого заданы параметрами x и y, определяющий область закрашивания, имеет цвет, совпадающий с crColor при режиме закрашивания FLOODFILLBORDER, или имеет цвет, не совпадающий с crColor, при режиме закрашивания FLOODFILLSURFACE, или если пиксель находится вне области отсечения.

Параметр crColor определяет либо цвет границы области, либо цвет самой области, что зависит от значения режима определения области закрашивания, задаваемого параметром nFillType. Всего возможны два режима определения области закрашивания:

FLOODFILLBORDER	область закрашивания определяется границей, цвет которой задается параметром crColor; этот режим обеспечивается также вызовом функции CDC::FloodFill
FLOODFILLSURFACE	область закрашивания определяется цветом, заданным параметром crColor; этот режим полезен для закрашивания областей, граничащих с областями разных цветов

Эта функция поддерживается только контекстами устройств в памяти и растровыми устройствами.

Прокрутка

Эта группа содержит только одну функцию — *ScrollDC*, которая обеспечивает прокрутку прямоугольных областей, как по вертикали, так и по горизонтали.

Управление выводом текста

Этот раздел познакомит вас с возможностями вывода текста, получения информации о выбранных шрифтах и т. п. Однако, как и в случае вывода графических объектов, знакомство будет неполным, т. к. в этом разделе мы практически не упоминаем о самих объектах рисования — шрифтах, о которых будет рассказано в следующей главе. Пока же познакомимся с функциями собственно вывода.

Вывод текста

Функции этой группы предназначены для организации вывода текстовых строк. Группа включает функции, от функций простейшего вывода и настройки режимов выравнивания до функций получения ряда характе-

ристик, таких как наименование шрифта (*GetTextFace*), длина заданной строки в логических единицах устройства вывода (*GetTextExtent* и *GetOutputTextExtent*) и установка режима выравнивания (*GetTextAlign*, *SetTextAlign* и *SetTextJustification*). Из всех функций вывода текста остановимся на основных и самых мощных.

Функция *ExtTextOut* позволяет осуществлять вывод текстовой строки в заданную прямоугольную область с использованием текущего шрифта и управляет режимами отображения фона. (Для вывода текста с заданной позиции используются функции *TextOut* и *TabbedTextOut*.) Функция представлена двумя версиями:

```
virtual BOOL CDC::ExtTextOut(
   int x,
   int y,
   UINT nOptions,
   LPCRECT lpRect,
   LPCTSTR lpszString,
   UINT nCount,
   LPINT lpDxWidths)
и
BOOL CDC::ExtTextOut(
   int x,
   int y,
   UINT nOptions,
   LPCRECT lpRect,
   const CString & str,
   LPINT lpDxWidths)
```

Основное отличие этих функций состоит в механизме задания строки символов. В первой версии функции строка задается параметром *lpszString*, а ее длина задается параметром *nCount*, если последний равен −1, то длина строки вычисляется. Во второй версии строка задается ссылкой на объект класса *CString*, задаваемый параметром *str*.

Параметр *lpDxWidths* указывает на массив чисел, определяющих расстояние между символами, если параметр равен NULL, то величина промежутков задается значением по умолчанию.

Параметры *x* и *y* задают координаты (в логических единицах) левого верхнего угла области вывода первого символа строки. Параметр *lpRect* задает прямоугольную область, в которой должен происходить вывод текста, и может быть равен NULL.

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

ETO_CLIPPED вывод текста осуществляется только в области, заданной параметром lpRect, а параметры x и y игнорируются

ETO_OPAQUE перед выводом текста прямоугольная область, заданная параметром lpRect, либо область, требуемая для вывода текста, заполняется текущим цветом фона

По умолчанию текущая позиция вывода после вызова функции не изменяется, однако, если задан флаг TA_UPDATECP при вызове функции *CDC::SetTextAlign*, параметры *x* и *y* будут проигнорированы, вывод будет осуществляться с текущей позиции, а по завершении вывода текущая позиция будет установлена либо в конец выведенной строки, либо установлена в соответствии с последним элементом массива (параметр *lpDxWidths*), в зависимости от того, какое значение больше.

Функция *DrawText* осуществляет текстовый вывод с различными способами форматирования:

```
virtual int CDC::DrawText(
   LPCTSTR lpszString,
   int nCount,
   LPRECT lpRect,
   UINT nFormat)
и
int CDC::DrawText(
   const CString & str,
   LPRECT lpRect,
   UINT nFormat) —
```

обе реализации в случае успеха возвращают высоту выведенного текста. Строка, как и в функции ExtTextOut, задается параметрами lpszString совместно с nCount или str. Параметр lpRect указывает на структуру *RECT* или объект класса CRect, которые содержат координаты (в логических единицах) прямоугольной области в которую собственно и происходит форматный вывод текста.

Параметр nFormat определяет способ форматирования заданного текста и может быть комбинацией следующих флагов:

DT_BOTTOM устанавливает выравнивание по нижней границе заданной прямоугольной области; должен комбинироваться с DT_SINGLELINE

DT_CALCRECT определение ширины и высоты прямоугольной области вывода; для текста в несколько строк используется ширина прямоугольной области, заданной параметром lpRect, и настраивается высота, так чтобы поместить весь текст; для текста в одну строку, напротив, ширина настраивается таким образом, чтобы точно соответствовать ширине выводимого текст; в любом случае функция возвращает высоту области, необходимую для полного вывода текста, однако собственно вывод *не осуществляется*; не может использоваться совместно с DT_TABSTOP

DT_CENTER устанавливает выравнивание по центру (по горизонтали) заданной области

DT_EXPANDTABS	при выводе производится замена символов табуляции на пробелы; каждая табуляция имеет размер в восемь символов
DT_EXTERNALLEADING	при определении высоты строки учитывается размер между строками; не может использоваться совместно с DT_TABSTOP
DT_LEFT	устанавливает выравнивание по левому краю заданной прямоугольной области
DT_NOCLIP	осуществляет вывод без учета ширины задаваемой прямоугольной области; не может использоваться совместно с DT_TABSTOP
DT_NOPREFIX	при выводе амперсант (&) не интерпретируется как указание подчеркивания следующего за ним символа, а двойной амперсант (&&) — как указание на вывод только одного; не может использоваться совместно с DT_TABSTOP
DT_RIGHT	устанавливает выравнивание по правому краю заданной прямоугольной области
DT_SINGLELINE	указание на вывод одной строки текста, при этом символы перевода строки и возврата каретки не разбивают строку
DT_TABSTOP	устанавливает размер табуляции, в старшем байте параметра nFormat содержится количество символов на табуляцию (по умолчанию размер табуляции равен восьми символам)
DT_TOP	устанавливает выравнивание по верхней границе заданной прямоугольной области; должен комбинироваться с DT_SINGLELINE
DT_VCENTER	устанавливает выравнивание по центру (по вертикали) заданной области; должен комбинироваться с DT_SINGLELINE
DT_WORDBREAK	устанавливает режим автоматического переноса строки между словами, если следующее слово выходит за рамки заданной прямоугольной области; символы переноса строки интерпретируются как обычно

Функция *DrawText* для вывода текста использует текущие шрифт, цвет символов и фона. Если флаг DT_NOCLIP не задан, текст не выходит за заданные границы. При этом в случае, если текущий шрифт слишком велик для заданной прямоугольной области, функция *DrawText* не делает попыток его уменьшить или заменить.

По умолчанию текущая позиция вывода после вызова функции не изменяется, однако, если задан флаг TA_UPDATECP при вызове функции

CDC::SetTextAlign, левый верхний угол области вывода будет проигнорирован — вывод будет осуществляться с текущей позиции, а по завершении вывода текущая позиция будет установлена в конец выведенного текста. При задании флага TA_UPDATECP флаг DT_WORDBREAK игнорируется.

К этой же группе относятся функции *GetOutputTextMetrics* и *GetTextMetrics*, которые обеспечивают приложение основной информацией о текущем шрифте контекста устройства графического вывода (*m_hDC*) и контекста, используемого для получения параметров вывода (*m_hAttribDC*), соответственно:

```
BOOL CDC::GetOutputTextMetrics(
  LPTEXTMETRIC lpMetrics)
```

и

```
BOOL CDC::GetTextMetrics(
  LPTEXTMETRIC lpMetrics)
```

Обе функции возвращают TRUE в случае успешного завершения и FALSE в противном случае. В качестве единственного параметра используется указатель на структуру *TEXTMETRIC*, которая и будет содержать, собственно, запрашиваемую информацию. Важность последней требует подробного описания структуры. Все размеры задаются в логических единицах, т. е. соответствуют текущей системе координат, устанавливаемой функцией *SetMapMode*.

```
typedef struct tagTEXTMETRIC {
    LONG tmHeight;
    LONG tmAscent;
    LONG tmDescent;
    LONG tmInternalLeading;
    LONG tmExternalLeading;
    LONG tmAveCharWidth;
    LONG tmMaxCharWidth;
    LONG tmWeight;
    LONG tmOverhang;
    LONG tmDigitizedAspectX;
    LONG tmDigitizedAspectY;
    BCHAR tmFirstChar;
    BCHAR tmLastChar;
    BCHAR tmDefaultChar;
    BCHAR tmBreakChar;
    BYTE tmItalic;
    BYTE tmUnderlined;
    BYTE tmStrickOut;
    BYTE tmPitchAndFamily;
    BYTE tmCharSet;
} TEXTMETRIC;
```

Остановимся подробнее на элементах структуры, часть которых представлена на рис. 3:

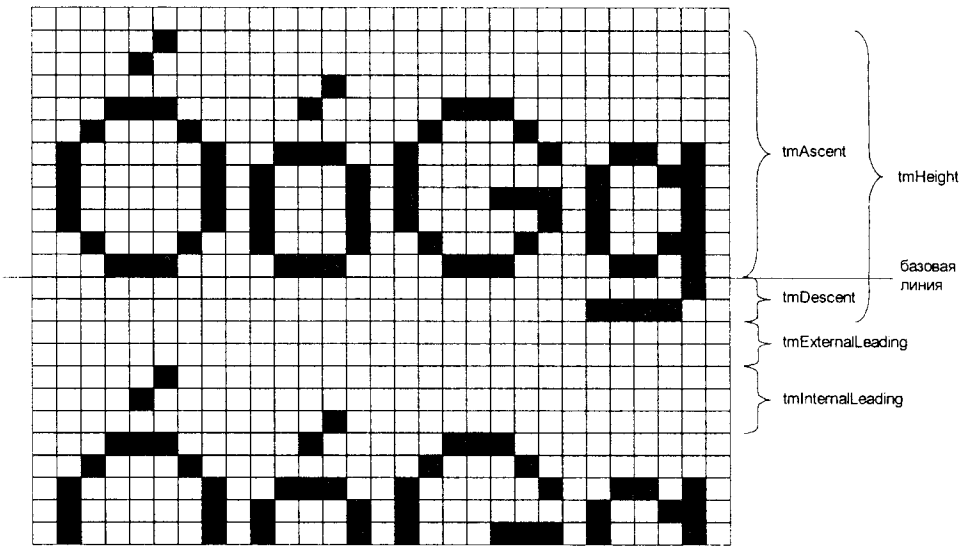


Рис. 3. Определение основных параметров шрифта с помощью элементов структуры TEXTMETRIC

tmHeight	высота символов (tmAscent + tmDescent)
tmAscent	высота области над базовой линией
tmDescent	размер области под базовой линией
tmInternalLeading	размер дополнительной области над символом, которая может быть использована для знаков ударения или диакритических символов[1]
tmExternalLeading	размер области между строками текста; при выводе эта область никак не заполняется ни в режиме OPAQUE (непрозрачный), ни в режиме TRANSPARENT (прозрачный)
tmAveCharWidth	средняя ширина символов; обычно определяется шириной для символа "x"; это значение не учитывает ширину символов в режимах **bold** (полужирный) и *italic* (курсив)
tmMaxCharWidth	максимальная ширина символов шрифта
tmWeight	"вес" или, что еще хуже, "жирность" символов шрифта
tmOverhang	значение, которое может быть добавлено для определения ширины строки для некоторых синтетических шрифтов (полужирных или курсивных)

[1] В русском языке также есть символы, использующие эту область, например, "Й", который выше "И" как раз на величину *tmInternalLeading*.

tmDigitizedAspectX, tmDigitizedAspectY	отношение tmDigitizedAspectX к tmDigitizedAspectY определяет масштаб устройства, для которого разрабатывался шрифт
tmFirstChar	первый определенный в шрифте символ
tmLastChar	последний определенный в шрифте символ
tmDefaultChar	символ, используемый для замены неопределенных символов
tmBreakChar	символ, используемый для разделения слов при выравнивании
tmItalic	ненулевое значение параметра указывает, что начертание текущего шрифта — курсивное
tmUnderlined	ненулевое значение параметра указывает, что текст, использующий текущий шрифт, будет подчеркнут
tmStrickOut	ненулевое значение параметра указывает, что текст, использующий текущий шрифт будет перечеркнут
tmPitchAndFamily	содержит флаги стиля шрифта. Старшие четыре бита определяют семейство шрифтов: FF_DECORATIVE, FF_DONTCARE, FF_MODERN, FF_ROMAN, FF_SCRIPT, FF_SWISS. Младшие четыре бита определяют тип шрифта: **TMPF_FIXED_PITCH** — определяет пропорциональность шрифта: если бит установлен, шрифт — пропорциональный, если сброшен — все символы шрифта имеют одинаковую ширину (моноширинный шрифт) **TMPF_VECTOR** — определяет, является ли шрифт векторным **TMPF_TRUETYPE** — если этот бит установлен, шрифт является шрифтом TrueType **TMPF_DEVICE** — если этот бит установлен, шрифт является шрифтом устройства
tmCharSet	определяет таблицу символов шрифта; Windows определяет следующие кодировки: ANSI_CHARSET, DEFAULT_CHARSET, SYMBOL_CHARSET, SHIFTJIS_CHARSET, GB2312_CHARSET, HANGEUL_CHARSET, CHINESEBIG5_CHARSET, OEM_CHARSET, JOHAB_CHARSET, HEBREW_CHARSET, ARABIC_CHARSET, GREEK_CHARSET, TURKISH_CHARSET, THAI_CHARSET, EASTEUROPE_CHARSET, RUSSIAN_CHARSET, MAC_CHARSET, BALTIC_CHARSET

Информация о шрифтах

Функции этой группы позволяют получить полную информацию о шрифтах (*GetFontData*, *GetKerningPairs*, *GetOutlineTextMetrics*, *GetGlyphOutline*, *GetCharABCWidth*, *GetCharWidth* и *GetOutputCharWidth*), позволяющую выби-

рать подходящий шрифт и точно определять параметры вывода, такие как размеры, область вывода и т. п.

Дополнительные функции

В этом разделе указаны группы функций, которые непосредственно не занимаются выводом или настройкой, однако связаны с контекстом устройства.

Интерфейс низкого уровня с устройствами

Класс *CDC* имеет в своем составе функции, которые могут напрямую взаимодействовать не только с принтерами, но и с графическими устройствами (видео-картами). Такие функции (*Escape* и *DrawEscape* соответственно) дают возможность программе вызывать выполнение действий, которые не реализованы средствами GDI. Естественно, наличие таких функций отчасти противоречит идеологии аппаратной независимости, но, тем не менее, они есть.

Управление процессом печати документов

Функции этой группы обеспечивают приложение возможностью взаимодействия с системным диспетчером печати (*StartDoc*, *StartPage*, *EndPage*, *EndDoc* и т. д.).

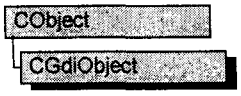
Метафайлы

Функции этой группы предназначены для отображения содержимого метафайла (функция *PlayMetaFile*) и добавления комментария к метафайлу (функция *AddMetaFileComment*).

Глава 18

Классы графических объектов

Класс CGdiObject



Класс *CGdiObject* является базовым для всех классов, представляющих и обеспечивающих интерфейс с графическими объектами Windows. Напомним, что таких объектов шесть — карандаш, кисть, битовый массив, палитра, шрифт и регион. Несмотря на то, что класс *CGdiObject* не является абстрактным, объекты этого класса создавать нет смысла. Значение этого класса в том, что он имеет набор функций, общих для всех производных классов.

Для получения представления о возможностях, предоставляемых классом, ознакомимся с его членами.

Данные класса

Для хранения дескриптора одного из шести типов объектов Windows, связанного с объектом класса, используется член класса *m_hObject*:

HANDLE CGdiObject::**m_hObject**

Функции-члены класса

Несмотря на то, что *m_hObject* является общедоступным членом, класс *CGdiObject* имеет в своем составе функцию, возвращающую копию дескриптора:

HGDIOBJ CGdiObject::**GetSafeHandle**()

Достоинства использования этой функции в том, что она возвращает значение NULL в случае, если самого объекта *CGdiObject* не существует, и указа-

тель *this* равен NULL. Любая иная проверка, подразумевающая доступ к членам класса, в таком случае приведет к ошибке доступа к памяти:[1]

```
CGdiObject *pObject = NULL;

if (pObject->m_hObject != NULL)   // в этом месте обращение
                                  // по несуществующему адресу -
                                  // ошибка доступа к памяти
  ...

if (pObject->GetSafeHandle() != NULL) // ошибки нет, т. к.
                                      // фактически доступа к
                                      // данным-членам класса нет
  ...
```

Аналогично классу *CDC,* класс *CGdiObject* имеет в своем составе статическую функцию, задача которой состоит в том, чтобы получить указатель на объект одного из классов графических объектов, созданный на основе заданного дескриптора:

static CGdiObject* PASCAL CGdiObject::**FromHandle**(
  HGDIOBJ hObject)

Скорее всего, указатель будет указывать на временный объект. Действия над временными графическими объектами аналогичны действиям над временными объектами классов контекстов устройств (см. предыдущую главу).

Созданный ранее графический объект Windows можно явно удалить, т. е. освободить соответствующие системные ресурсы, для чего необходимо воспользоваться функцией *DeleteObject.*

BOOL CGdiObject::**DeleteObject**()

Деструктор класса в любом случае удалит графический объект Windows как раз при помощи этой функции. Однако следует понимать, что этот объект не может быть текущим для контекста устройства.

Фактически вызов функции *DeleteObject* может понадобиться только в одном случае — если один и тот же объект, созданный на основе класса *CGdiObject*, необходим для обеспечения интерфейса с различными графическими объектами Windows. Поясним на примере.

Функции *OnPaint*, классов *CNeptuneFrame* и *CTitanFrame* рисуют квадрат, причем для горизонтальных и вертикальных его сторон используются различные по толщине и цвету карандаши.

В первом случае для каждого типа сторон создаются различные объекты класса *CPen — pen_1* и *pen_2.*

---

[1] Напомним, что и в Windows 95 и в Windows NT младшие адреса виртуального адресного пространства зарезервированы для обнаружения работы с нулевыми и близкими к нулю указателями на ранней стадии отладки, т. к. такие действия ошибочны.

```
void CNeptuneFrame::OnPaint()
{
  CPaintDC dc(this);

  CPen pen_1(PS_SOLID, 1, RGB(0, 0, 0));
  CPen *pPen = dc.SelectObject(&pen_1);

  dc.MoveTo(100, 100);
  dc.LineTo(200, 100);
  dc.MoveTo(100, 200);
  dc.LineTo(200, 200);

  CPen pen_2(PS_SOLID, 2, RGB(192, 192, 192));

  // заменяем первый карандаш вторым
  dc.SelectObject(&pen_2);

  // рисуем при помощи второго карандаша,
  // но первый еще занимает ресурсы

  dc.MoveTo(100, 100);
  dc.LineTo(100, 200);
  dc.MoveTo(200, 100);
  dc.LineTo(200, 200);

  // естественно, заменяем наш карандаш,
  // тем, что был установлен ранее
  dc.SelectObject(pPen);
}
```

Во втором случае мы обошлись только одним объектом класса *CPen* для каждого типа сторон.

```
void CTitanFrame::OnPaint()
{
  CPen *pPen, pen;

  pen.CreatePen(PS_SOLID, 1, RGB(0, 0, 0));
  pPen = dc.SelectObject(&pen);

  dc.MoveTo(100, 100);
  dc.LineTo(200, 100);
  dc.MoveTo(100, 200);
  dc.LineTo(200, 200);

  dc.SelectObject(pPen);
  pen.DeleteObject();

  // теперь программа не занимает ресурсы для хранения карандаша
  pen.CreatePen(PS_SOLID, 2, RGB(192, 192, 192));
  dc.SelectObject(&pen);

  dc.MoveTo(100, 100);
  dc.LineTo(100, 200);
```

```
  dc.MoveTo(200, 100);
  dc.LineTo(200, 200);

  // естественно, заменяем наш карандаш,
  // тем, что был установлен ранее
  dc.SelectObject(pPen);
}
```

Обратите внимание на то, что перед вызовом функции *DeleteObject* мы восстановили в контексте устройства ранее установленный карандаш. Связано это с тем, что по понятным причинам нельзя удалять графический объект в тот момент, когда он является текущим для контекста устройства. Простое повторное создание нового объекта библиотекой запрещено. Функция *CGdiObject::Attach*, осуществляющая связывание объекта Windows с объектом соответствующего графического класса, не берет на себя ответственность за удаление ранее созданных графических объектов Windows. Вместо этого она сперва проверяет наличие таких объектов и, при обнаружении, генерирует ошибку. Вот исходный код (и соответствующий комментарий), осуществляющий эти действия:

```
BOOL CGdiObject::Attach(HGDIOBJ hObject)
{
  ASSERT(m_hObject == NULL);  // only attach once,
                              // detach on destroy
  ...
}
```

Также обратите внимание, что повторно функция *DeleteObject* не вызывается — она используется в деструкторе, который автоматически вызывается по выходу из области видимости, в данном случае по завершении функции *OnPaint*. Интересно, что после выхода из области видимости объект класса уже не существует, а вот графический объект может быть фактически не удален. Чтобы этого не случилось следует обязательно заменять пользовательские объекты в контексте устройства, иначе объект Windows удален не будет даже после вызова функции *DeleteObject*, что приведет к постепенному исчерпанию ресурсов и краху системы.

Класс *CGdiObject* имеет в своем составе функцию, которая создает объекты, связанные с предопределенными графическими объектами Windows:

BOOL CGdiObject::**CreateStockObject**(int nIndex)

Параметр nIndex принимает значения, аналогичные тем, что используются функцией *CDC::SelectStockObject*, описанной в предыдущей главе.

И, наконец, последняя функция *GetObject* позволяет получить информацию о графическом объекте, доступ к которому может быть не реализован функциями-членами соответствующих классов библиотеки MFC.

int CGdiObject::**GetObject**(
int nCount,
LPVOID lpObject)

Параметр lpObject является указателем на блок данных, длина которого задается параметром nCount. Функция возвращает количество байт информации, фактически переданной приложению, и 0 в случае возникновения ошибки. Для каждого графического класса, кроме CRgn, предусмотрен доступ к перечисленным ниже структурам:

Класс	Тип данных
CPen	LOGPEN
CBrush	LOGBRUSH
CFont	LOGFONT
CBitmap	BITMAP
CPalette	int — количество элементов палитры

Например, для битовых массивов использование этой функции может быть единственным способом получить информацию о размерах, т. к. соответствующая функция *CBitmap::GetBitmapDimension* возвращает размеры только в том случае, если они были установлены функцией *CBitmap::SetBitmapDimension*.

В заключение этого раздела необходимо отметить, что у всех классов, инкапсулирующих графические объекты Windows, есть функции-члены, которые являются перегруженными операторами, приводящими соответствующий объект класса к дескриптору объекта Windows, упрощая использование функций Win32 API:

CGdiObject::**operator HGDIOBJ**()

CPen::**operator HPEN**()

CBrush::**operator HBRUSH**()

CFont::**operator HFONT**()

CBitmap::**operator HBITMAP**()

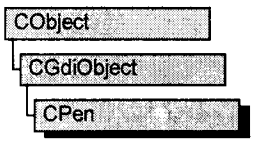
CPalette::**operator HPALETTE**()

CRgn::**operator HRGN**()

Далее мы подробно остановимся на пяти из шести классов, инкапсулирующих возможности графических объектов Windows.[1]

[1] Класс *CPalette* нами рассматриваться не будет из-за его менее широкого применения, прежде всего, в специальных областях обработки изображений, а также при ограниченных цветовых возможностях устройств вывода.

Карандаш (класс CPen)



Класс *CPen* инкапсулирует графический объект Windows — "карандаш". Карандаши используются для рисования различных линий и контуров объектов. Эти объекты имеют наиболее богатые возможности их создания — в Windows имеется целый набор различных предопределенных стилей, свойств и типов, которые открывают возможность простого и эффективного использования карандашей и, соответственно, рисования самых разнообразных линий.

Как и все другие классы графических объектов, класс *CPen* позволяет создавать объекты класса, не связывая их при этом с объектами Windows, или наоборот — при создании объекта класса непосредственно создавать и соответствующий графический объект.[1]

Что касается последней возможности, то она дублируется наличием функций создания карандаша Windows уже после создания объекта класса. О возможных способах диагностики результатов создания мы говорили в главе 16. Так как собственно создание объекта Windows заключается в формировании соответствующих данных, которые передаются системе функциями-членами класса *CPen*, мы не будем дублировать описание функций создания, а подробно остановимся только на конструкторах.

CPen::**CPen**() —
конструктор без параметров, позволяет создавать объекты класса CPen, никак не связанные с объектами Windows.

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

и

```
CPen::CPen(
  int nPenStyle,
  int nWidth,
  const LOGBRUSH* pLogBrush,
  int nStyleCount = 0,
  const DWORD* lpStyle = NULL) —
```

являются конструкторами, позволяющими при создании объекта класса CPen создавать карандаш Windows. Наличие двух версий конструкторов, создающих объект Windows, объясняется различием создаваемых карандашей и возможными значениями прежде всего параметра nPenStyle.

Далее мы рассмотрим отдельно для каждой версии конструктора возможные значения параметра *nPenStyle*.

---

[1] Последняя возможность реализована пока только для класса *CBrush*.

В результате использования конструктора первой версии возможно создание карандашей с предопределенными в Windows стилями для рисования следующих типов линий:

PS_SOLID	сплошная линия
PS_DASH	длинные штрихи, толщина которых равна 1
PS_DOT	короткие штрихи, толщина которых равна 1
PS_DASHDOT	штрих-пунктир, толщина которого равна 1
PS_DASHDOTDOT	двойной штрих-пунктир, толщина которого равна 1
PS_NULL	прозрачная линия
PS_INSIDEFRAME	сплошная линия для рисования по внутренней границе замкнутой области, например, при использовании функций класса CDC — Rectangle, RoundRect, Ellipse, Chord и т. д.; при использовании карандаша данного типа для рисования линий, например, при помощи функции LineTo, результат будет тем же, что и для стиля PS_SOLID; еще одна область применения этого стиля — создание карандашей толщиной более одного пикселя и имеющих цвет, недоступный для текущего устройства; в этом случае линии будут состоять из пикселей доступных цветов, совокупность которых наиболее близко будет соответствовать заданному цвету

Для наглядности продемонстрируем линии различных стилей. Обратите внимание, что при выводе линий одной длины фактически некоторые штриховые линии оканчиваются раньше сплошных (рис. 4), что вполне естественно — эти линии завершились в месте пустого пространства между штрихами. Такой эффект может негативно сказаться при решении конкретных задач.

```
void CLineStyleFrame::OnPaint()
{
  CPaintDC dc(this);

  struct { int m_nStyle; LPTSTR m_csStyle; } lnStyle[] = {
    PS_SOLID,         _T("PS_SOLID"),
    PS_DASH,          _T("PS_DASH"),
    PS_DOT,           _T("PS_DOT"),
    PS_DASHDOT,       _T("PS_DASHDOT"),
    PS_DASHDOTDOT,    _T("PS_DASHDOTDOT"),
    PS_NULL,          _T("PS_NULL"),
    PS_INSIDEFRAME,   _T("PS_INSIDEFRAME")
  };

  //
  // чтобы при любом размере шрифта тексты не пересекались,
  // получим и учтем характеристики текущего шрифта
  //
  TEXTMETRIC tm;
```

```
  dc.GetTextMetrics(&tm);

  // устанавливаем режим "прозрачного" вывода
  dc.SetBkMode(TRANSPARENT);

  for (int i = 0; i < sizeof(lnStyle) / sizeof(lnStyle[0]); i++)
  {
    // будем рисовать разными по стилю, но имеющими
    // единичную толщину и черный цвет карандашами
    CPen pen(lnStyle[i].m_nStyle, 1, RGB(0, 0, 0));
    CPen *pPen = dc.SelectObject(&pen);

    // учитываем высоту символов и расстояние между строками
    int nY = 10+i*(tm.tmHeight+tm.tmExternalLeading);

    dc.TextOut(10, nY, lnStyle[i].m_csStyle);

    dc.MoveTo(200, nY+(tm.tmHeight+tm.tmExternalLeading) / 2);
    dc.LineTo(400, nY+(tm.tmHeight+tm.tmExternalLeading) / 2);

    dc.SelectObject(pPen);
  }
}
```

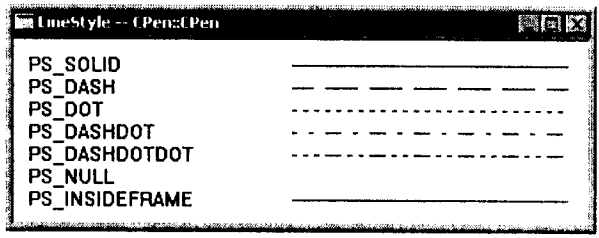


Рис. 4. Результат работы функции CLineStyleFrame::OnPaint

Использование конструктора второй версии позволяет комбинировать в параметре *nPenStyle* стиль, тип, режим рисования концов линии и режим соединения линий. Каждый вид параметров может быть представлен только одним значением, а заданные значения комбинируются при помощи оператора OR (|).

Карандаши Windows могут быть двух типов — PS_GEOMETRIC и PS_COSMETIC. Отличие типов заключается в использовании кистей для рисования линий и требований к заданию ширины. Кроме того, от выбранного типа зависит интерпретация параметров задания размеров штрихов для пользовательских стилей.

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

Для карандашей также возможно задание режима рисования концов (естественно, это имеет смысл при рисовании широких линий), их три, для первых двух режимов фактическая длина линии может быть больше заданной:

PS_ENDCAP_ROUND концы линий закруглены

PS_ENDCAP_SQUARE концы линий имеют прямоугольную форму

PS_ENDCAP_FLAT то же, что и для PS_ENCAP_SQUARE, но концы линии не выходят за заданные размеры.

Для соединения линий можно выбирать один из следующих режимов:

PS_JOIN_BEVEL соединение обрезано

PS_JOIN_MITER соединение определяется продолжением контура линий, если только расстояние между концами линий не превышает заданную функцией *CDC::SetMiterLimit* величину, в противном случае соединение обрезается, как и в случае задания режима PS_JOIN_BEVEL

PS_JOIN_ROUND соединение закруглено

Одно из требований по использованию режима рисования концов линий и соединений заключается в том, что рисование линий при помощи карандашей, созданных с флагами PS_ENDCAP_SQUARE, PS_ENDCAP_FLAT, PS_JOIN_BEVEL и PS_JOIN_MITER, можно осуществлять только при помощи механизма использования контуров, что и продемонстрировано в примере. Обратите внимание, что по оси каждой линии, имеющей тип PS_GEOMETRIC, для наглядности проведены сплошные тонкие линии с теми же координатами.

```
void CCapJoinFrame::OnPaint()
{
  CPaintDC dc(this);

  int nEnd[] =
    { PS_ENDCAP_ROUND, PS_ENDCAP_SQUARE, PS_ENDCAP_FLAT };

  int nJoin[] =
    { PS_JOIN_BEVEL, PS_JOIN_MITER, PS_JOIN_ROUND };

  CString csEnd[] = { _T("PS_ENDCAP_ROUND"),
    _T("PS_ENDCAP_SQUARE"), _T("PS_ENDCAP_FLAT") };

  CString csJoin[] = { _T("PS_JOIN_BEVEL"),
    _T("PS_JOIN_MITER"), _T("PS_JOIN_ROUND") };

  //
  // создаем новый шрифт и заменяем им текущий,
  // т. к. он слишком велик для вывода имен режимов
  //
```

```
CFont font;
font.CreateFont(12, 0, 0, 0, FW_BOLD, 0, 0, 0, DEFAULT_CHARSET,
  OUT_DEFAULT_PRECIS, CLIP_DEFAULT_PRECIS, PROOF_QUALITY,
  VARIABLE_PITCH | FF_DONTCARE, "Arial");

CFont *pFont = dc.SelectObject(&font);

// устанавливаем режим "прозрачного" вывода
dc.SetBkMode(TRANSPARENT);

// перебираем все возможные комбинации (их 9)
for (int i = 0; i < 3; i++)
{
  dc.TextOut(100+i*120, 10, csEnd[i]);
  dc.TextOut(10, 90+i*80, csJoin[i]);

  for (int j = 0; j < 3; j++)
  {
    //
    // для создания карандашей понадобится структура LOGBRUSH,
    // которая определена в Win32 API для описания кистей
    //

    LOGBRUSH lb;

    lb.lbStyle = BS_SOLID;
    lb.lbColor = RGB(128, 128, 128);
    lb.lbHatch = 0;

    CPen pen(
      PS_SOLID | PS_GEOMETRIC | nEnd[i] | nJoin[j],
      20, &lb);
    CPen *pPen = dc.SelectObject(&pen);

    //
    // чтобы увидеть различие режимов, придется создавать
    // контуры и только затем их обводить
    //

    dc.BeginPath();

    dc.MoveTo(100+i*120,  50+j*80);
    dc.LineTo(140+i*120, 150+j*80);
    dc.LineTo(180+i*120,  50+j*80);

    dc.EndPath();

    dc.StrokePath();

    // для рисования базовых линий используем
    // карандаш, заданный по умолчанию
    dc.SelectObject(pPen);

    dc.MoveTo(100+i*120,  50+j*80);
```

```
      dc.LineTo(140+i*120, 150+j*80);
      dc.LineTo(180+i*120,  50+j*80);
    }
  }

  // шрифт, также как и другие графические объекты,
  // надо восстановить в контексте
  dc.SelectObject(pFont);
}
```

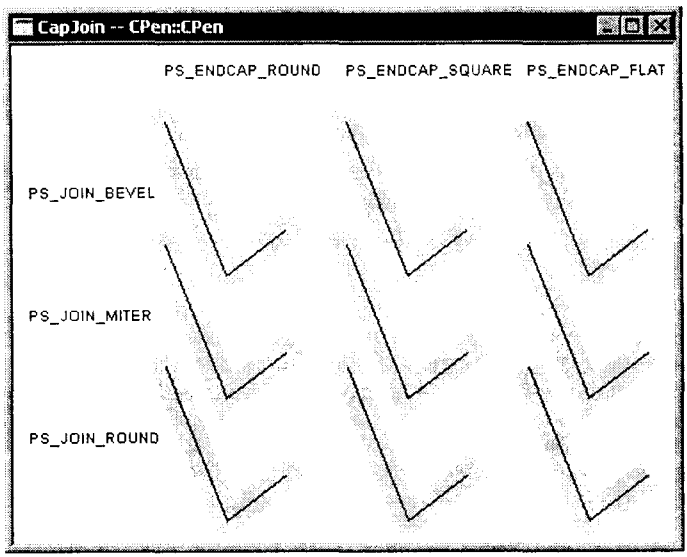


Рис. 5. Результат работы функции CCapJoinFrame::OnPaint

При создании карандаша необходимо также задать ширину при помощи параметра *nWidth*. При задании в первой версии конструктора ширины, равной 0, гарантируется вывод линии шириной в один пиксель независимо от режима вывода (mapping mode). При использовании второй версии конструктора для карандашей типа PS_GEOMETRIC ширина задается в логических единицах, а для карандашей типа PS_COSMETIC ширина должна быть равна 1.

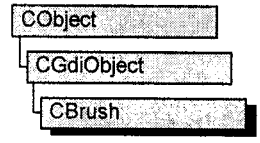
Для задания цвета карандаша используется параметр *crColor*.

Параметр *pLogBrush* является указателем на структуру *LOGBRUSH* для создания кисти, с помощью которой будет производиться рисование линий. Для карандашей типа PS_COSMETIC элемент структуры *LOGBRUSH*::*lbStyle* должен быть равен BS_SOLID.

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

Как упоминалось в начале раздела, для создания карандашей могут быть использованы функции *CreatePen*, *CreatePenIndirect* или *CreateStockObject*. При помощи функции *CreatePenIndirect* возможно создание только обычных карандашей, т. е. таких же, какие можно создавать при помощи первой версии конструктора, описанной выше.

Кисть (класс CBrush)



Класс *CBrush* инкапсулирует графический объект Windows — "кисть" для использования, прежде всего, в качестве основы при закрашивании замкнутых фигур. Кисти могут быть однотонные, со штриховкой или созданные на основе шаблона.

Как и все другие классы графических объектов, класс *CBrush* имеет в своем составе конструктор без параметров:

CBrush::**CBrush**()

который создает объект класса, но для его использования требует вызова одной из "создающих" функций — *CreateSolidBrush*, *CreateHatchBrush*, *CreateBrushIndirect*, *CreatePatternBrush*, *CreateDIBPatternBrush*, *CreateSysColorBrush*. Первые четыре функции полностью дублируют возможности конструкторов. Пятая функция создает кисть на основе битового массива, но, в отличие от четвертой функции, этот массив хранит изображение в независимом от устройств виде. Последняя функция создает кисть со штриховкой на основе системного цвета.

Кроме конструктора по умолчанию, класс *CBrush* имеет в своем составе конструкторы, позволяющие создавать кисти одного цвета (однотонные):

```
CBrush::CBrush(COLORREF crColor)
```

кисти со штриховкой:

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

и кисти на основе битового массива, используемого в качестве шаблона:

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

Параметр *crColor* определяет цвет закрашивания для кистей одного цвета, а для кистей со штриховкой этот параметр определяет цвет только штриховки.

Чтобы создать кисть со штриховкой, необходимо установить параметр *nIndex* в одно из следующих значений:

HS_BDIAGONAL	45-градусная штриховка слева направо и сверху вниз
HS_CROSS	горизонтальная и вертикальная штриховка крест накрест
HS_DIAGCROSS	45-градусная штриховка крест накрест

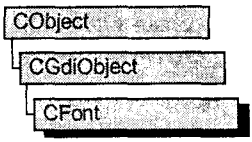
HS_FDIAGONAL 45-градусная штриховка слева направо и снизу вверх

HS_HORIZONTAL горизонтальная штриховка

HS_VERTICAL вертикальная штриховка

Для создания кисти на основе битового массива необходимо сначала создать или загрузить битовый массив, а затем, передав указатель на него, вызвать последний из перечисленных конструкторов. При этом надо иметь в виду, что битовый массив должен иметь размеры не меньшие чем 8 × 8 пикселей, а сама кисть будет создана на основе самой левой верхней квадратной области битового массива размером также 8 × 8 пикселей.

Шрифт (класс CFont)



Класс *CFont* инкапсулирует графический объект Windows — "шрифт". Для создания объекта этого класса имеется конструктор без параметров, однако чтобы создать собственно шрифт, необходимо вызвать одну из следующих функций: *CreateFont*, *CreateFontIndirect*, *CreatePointFont* или *CreatePointFontIndirect*. Для использования созданного шрифта необходимо установить его в конкретный контекст устройства, а по завершении использования заменить ранее установленным шрифтом — все как и для других графических объектов.

Многие классы, инкапсулирующие графические объекты Windows, имеют в своем составе функции создания объектов при помощи структур. Однако мы описывали в основном те функции, которые принимают в качестве параметров те же значения, что и поля таких структур. В случае со шрифтами мы поступим иначе прежде всего потому, что структура *LOGFONT*, используемая для создания шрифта, достаточно часто используется не только вместе с классом *CFont*, а также, например, и при выборе шрифта с помощью блока диалога, созданного при помощи класса *CFontDialog* (см. 15-ю главу 1-го тома книги).

Итак, для создания шрифта используется функция *CreateFontInderect*. Вот ее описание:

BOOL CFont::**CreateFontIndirect**(
  const LOGFONT* lpLogFont)

где параметр lpLogFont указывает на структуру *LOGFONT*, которая, в свою очередь, определяет шрифт, необходимый приложению.

Остановимся более подробно на составе структуры и описании ее полей:

```
typedef struct tagLOGFONT {
    LONG lfHeight;
    LONG lfWidth;
    LONG lfEscapement;
```

```
    LONG lfOrientation;
    LONG lfWeight;
    BYTE lfItalic;
    BYTE lfUnderline;
    BYTE lfStrikeOut;
    BYTE lfCharSet;
    BYTE lfOutPrecision;
    BYTE lfClipPrecision;
    BYTE lfQuality;
    BYTE lfPitchAndFamily;
    TCHAR lfFaceName[LF_FACESIZE];
} LOGFONT;
```

Поле *lfHeight* определяет высоту (в логических единицах) знакомест или символов шрифта. Высота последних определяется как разность между полной высотой знакоместа и размером области для отображения специальных знаков, например, ударения.[1] Это поле структуры *LOGFONT* может интерпретироваться, в зависимости от знака, следующим образом:

> 0	значение преобразуется в единицы устройства вывода и используется для выбора шрифта из имеющихся, соответствующего по высоте знакоместа
0	для нахождения шрифта используется значение высоты по умолчанию
< 0	значение преобразуется в единицы устройства вывода и используется для выбора шрифта из имеющихся, соответствующего по высоте символа

При поиске подходящего шрифта выбирается шрифт с наиболее подходящей высотой знакоместа или символа (высота должна быть меньше или равна запрашиваемой).

Для режима отображения MM_TEXT высота может быть определена через размер шрифта, задаваемого в точках (points), следующим образом:

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

Поле *lfWidth* определяет среднюю ширину символов в логических единицах. Нулевое значение поля *lfWidth* обеспечивает выбор шрифта из имеющихся, наиболее подходящего к конкретному устройству по возможностям масштабирования.

Поле *lfEscapement* определяет угол (в десятых долях градуса) между базовой линией выводимого текста и горизонталью.

Поле *lfOrientation* определяет угол (в десятых долях градуса) между базовой линией каждого выводимого символа и горизонталью.

К сожалению, в рамках Windows 95 значения обоих полей *lfEscapement* и *lfOrientation* должны быть равны, т. е. существует возможность вывода под

[1] Более подробно о терминах, определяющих размеры символов, см. главу 17.

наклоном только текста всей строки, когда базовые линии символов лежат на базовой линии строки. В Windows NT при установке графического режима GM_ADVANCED, устанавливаемого функцией Win32 API ::*SetGraphicsMode*, возможности по выводу текста расширяются.

Поле *lfWeight* определяет толщину линий начертания символов и может принимать значения от 0 до 1000. При нулевом значении используется значение по умолчанию. Некоторым значениям даны символьные наименования:

FW_DONTCARE	0
FW_THIN	100
FW_EXTRALIGHT	200
FW_ULTRALIGHT	200
FW_LIGHT	300
FW_NORMAL	400
FW_REGULAR	400
FW_MEDIUM	500
FW_SEMIBOLD	600
FW_DEMIBOLD	600
FW_BOLD	700
FW_EXTRABOLD	800
FW_ULTRABOLD	800
FW_HEAVY	900
FW_BLACK	900

Поле *lfItalic* определяет курсивное начертание шрифта.

Поле *lfUnderline* определяет подчеркнутое начертание шрифта.

Поле *lfStrikeOut* определяет перечеркнутое начертание шрифта.

Поле *lfCharSet* определяет таблицу кодировки шрифта. Определены следующие таблицы: ANSI_CHARSET, DEFAULT_CHARSET, SYMBOL_CHARSET, SHIFTJIS_CHARSET, GB2312_CHARSET, HANGEUL_CHARSET, CHINESEBIG5_CHARSET, OEM_CHARSET.

В рамках Windows 95 дополнительно определены следующие таблицы: JOHAB_CHARSET, HEBREW_CHARSET, ARABIC_CHARSET, GREEK_CHARSET, TURKISH_CHARSET, THAI_CHARSET, EASTEUROPE_CHARSET, RUSSIAN_CHARSET, MAC_CHARSET, BALTIC_CHARSET.

Значение OEM_CHARSET определяет таблицу кодировки конкретной версии операционной системы. Значение DEFAULT_CHARSET для идентификации шрифта требует задания имени (поле *lfFaceName*) и размера, однако отсутствие запрашиваемого шрифта может привести к непредсказуе-

мым результатам, т. к. он будет подменен шрифтом, подходящим по другим параметрам, но неизвестной заранее кодировки.

Особенно важно задание правильной таблицы кодировки при задании в явном виде имени шрифта, т. к. именно кодировка в **первую очередь определяет набор установленных шрифтов**, среди которых будет произведен поиск по имени (рис. 6):

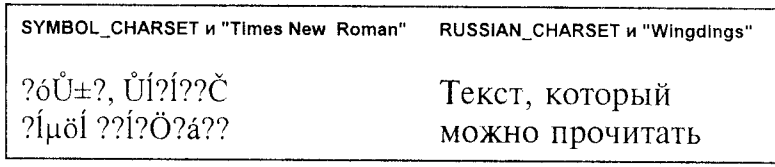


Рис. 6. Пример доминирования заданной таблицы кодировки над именем шрифта

Поле *lfOutPrecision* определяет механизм выбора шрифта, т. е. насколько шрифт должен соответствовать заданным параметрам, прежде всего типу шрифта (растровый, системный или шрифт TrueType), а также ширине, высоте, ориентации и т. д. Это поле может принимать одно из следующих значений:

OUT_CHARACTER_PRECIS	не используется
OUT_DEFAULT_PRECIS	определяет механизм выбора, принятый по умолчанию
OUT_DEVICE_PRECIS	указывает на то, чтобы в случае, когда в системе установлены несколько шрифтов с одним и тем же названием, был выбран системный шрифт
OUT_OUTLINE_PRECIS	только для Windows NT: выбор осуществляется из набора векторных или шрифтов TrueType
OUT_RASTER_PRECIS	указывает на то, чтобы в случае, когда в системе установлены несколько шрифтов с одним и тем же названием, был выбран растровый шрифт
OUT_STRING_PRECIS	используется для получения списка растровых шрифтов
OUT_STROKE_PRECIS	используется для получения списка векторных и шрифтов TrueType
OUT_TT_ONLY_PRECIS	задает выбор только шрифта TrueType, и если ни один такой шрифт не установлен, используется механизм выбора, принятый по умолчанию
OUT_TT_PRECIS	указывает на то, чтобы в случае, когда в системе установлены несколько шрифтов с одним и тем же названием, был выбран шрифт TrueType

Поле *lfClipPrecision* определяет механизм отсечения символов шрифта в случае, если символы частично оказываются вне области вывода. Это поле может принимать одно или несколько из следующих значений:

CLIP_DEFAULT_PRECIS	определяет использование механизма, принятого по умолчанию
CLIP_CHARACTER_PRECIS	не используется
CLIP_STROKE_PRECIS	используется только для получения списка растровых, векторных или шрифтов TrueType
CLIP_MASK	не используется
CLIP_EMBEDDED	необходимо указывать этот флаг при использовании внедренных шрифтов
CLIP_LH_ANGLES	при использовании этого флага отсчет направления угла поворота (по часовой стрелки или против) зависит от выбранной системы координат; если флаг не используется, то поворот текста, выводимого с использованием системных шрифтов, производится против часовой стрелки, однако для всех остальных шрифтов направление поворота определяется системой координат
CLIP_TT_ALWAYS	не используется

Поле *lfQuality* определяет желаемое качество вывода, которое зависит от выбранного шрифта. Определены следующие три значения качества вывода:

DEFAULT_QUALITY	качество вывода текста не критично
DRAFT_QUALITY	качество вывода текста менее важно, чем для PROOF_QUALITY; для растровых шрифтов разрешено масштабирование; при необходимости возможно создание синтетических начертаний шрифтов — полужирного, курсивного, подчеркнутого и перечеркнутого
PROOF_QUALITY	качество вывода текста более важно, чем соответствие заданным атрибутам; для растровых шрифтов масштабирование запрещено, выбирается шрифт наиболее подходящий по размеру; при необходимости возможно создание синтетических начертаний шрифтов — полужирного, курсивного, подчеркнутого и перечеркнутого

Поле *lfPitchAndFamily* определяет тип и семейство шрифтов. Это поле позволяет выбрать шрифт, наиболее похожий на заданный по имени в случае, если последний не установлен в системе. Для задания типа и семейства шрифта используется операция OR (|) между определениями типа и семейства соответственно.

Тип шрифта определяется двумя младшими битами поля и может принимать одно из следующих значений:

DEFAULT_PITCH	тип, принятый по умолчанию
FIXED_PITCH	непропорциональный шрифт, т. е. все символы шрифта имеют одинаковый размер
VARIABLE_PITCH	пропорциональный шрифт

Семейство определяется старшими четырьмя битами поля и может принимать одно из следующих значений:

FF_DECORATIVE	декоративный шрифт, например, Old English
FF_DONTCARE	семейство шрифта не имеет значения
FF_MODERN	непропорциональный шрифт с засечками или без них, например CourierNew®
FF_ROMAN	пропорциональный шрифт с засечками, например MS® Serif
FF_SCRIPT	шрифт, схожий по начертанию с рукописным, например Script или Cursive
FF_SWISS	пропорциональный шрифт без засечек, например MS® Sans Serif

Поле *lfFaceName* содержит имя шрифта. Длина имени не может быть более 32 символов, включая завершающий нуль-символ. Если имя не указано или шрифт с таким именем не найден, поиск подходящего шрифта будет проводиться с использованием других заданных параметров. Однако в ряде случаев имя не является определяющим (см. комментарии к полю *lfCharSet*).

Также для создания шрифта можно воспользоваться функцией *CreateFont*, которая принимает в качестве параметров те же значения, что и поля структуры *LOGFONT*:

```
BOOL CFont::CreateFont(
   int nHeight,
   int nWidth,
   int nEscapement,
   int nOrientation,
   int nWeight,
   BYTE bItalic,
   BYTE bUnderline,
   BYTE cStrikeOut,
   BYTE nCharSet,
   BYTE nOutPrecision,
   BYTE nClipPrecision,
   BYTE nQuality,
   BYTE nPitchAndFamily,
   LPCTSTR lpszFacename)
```

В ряде случаев при выборе шрифта нет необходимости столь подробно указывать его ожидаемые параметры. В то же время, использование большинства текстовых редакторов приучает к тому, что размер шрифта характе-

ризуется вполне точным, если хотите, абсолютным значением — размер задается в точках. А точки (points), как вы помните, вовсе не экранные точки, т. е. пиксели, и их размер известен — 1/72 дюйма. Ранее мы приводили формулу для вычисления размера в пикселях на основе размера в точках. Класс *CFont* имеет в своем составе функцию, которая, с одной стороны, создает шрифт на основе только его имени, а с другой — устанавливает желаемый размер как раз в привычных всем единицах:

```
BOOL CFont::CreatePointFont(
  int nPointSize,
  LPCTSTR lpszFaceName,
  CDC* pDC = NULL)
```

Первый параметр *nPointSize* определяет величину шрифта в десятых долях точки, например, при выборе шрифта с размером символов в 8 точек, необходимо установить значение этого параметра в 80.

Имя шрифта задается параметром *lpszFaceName*, нулевое значение которого приводит к выбору системно-независимого шрифта.

Последний параметр *pDC* нужен для правильного преобразования размера шрифта в логические единицы, которые определяются именно для контекста устройства. Нулевое значение параметра приводит к выбору в качестве контекста устройства контекст экрана и, соответственно, его логических единиц.

Аналог функции *CreateFontIndirect* — функция *CreatePointFontIndirect* — также в качестве параметра принимает указатель на структуру *LOGFONT*, однако единицами измерения поля *lfHeight* этой структуры являются точки.

В качестве примера приведем функцию, которая выводит строку "Visual C++" под разными углами:

```
void CCreateFontFrame::OnPaint()
{
  CPaintDC dc(this);

  CRect rect;
  GetClientRect(rect);

  CFont font;
  LOGFONT lf = {
    32, 0, 0, 0, FW_NORMAL, 0, 0, 0, DEFAULT_CHARSET,
    OUT_TT_ONLY_PRECIS, CLIP_DEFAULT_PRECIS, PROOF_QUALITY,
    VARIABLE_PITCH | FF_DONTCARE, NULL
  };

  // устанавливаем режим "прозрачного" вывода
  dc.SetBkMode(TRANSPARENT);

  for (int angle = 0; angle < 3600; angle += 450)
  {
```

```
    //
    // задаем угол наклона всей строки
    //
    lf.lfEscapement = angle;
    lf.lfOrientation = angle;

    font.CreateFontIndirect(&lf);

    // назначаем текущим ранее подготовленный шрифт
    CFont *pFont = dc.SelectObject(&font);

    dc.TextOut(rect.Width() / 2, rect.Height() / 2,
      _T("    Visual C++"));

    dc.SelectObject(pFont);
    font.DeleteObject();
  }
}
```

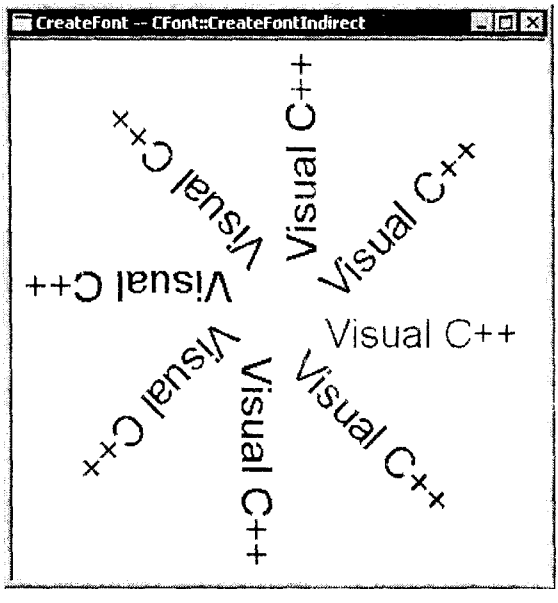
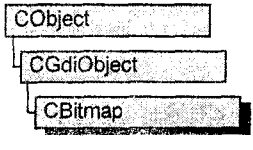


Рис. 7. Результат работы функции CCreateFontFrame::OnPaint

Битовый массив (класс CBitmap)



Класс *CBitmap* инкапсулирует графический объект Windows — "битовый массив". Для работы с объектом класса необходимо создать сам объект, затем создать собственно битовый массив (*CreateBitmap*, *CreateBitmapInderect*, *CreateCompatibleBitmap* или *CreateDiscardableBitmap*), в

том числе путем загрузки соответствующего ресурса (*LoadBitmap*, *LoadOEMBitmap* или *LoadMappedBitmap*), или установить связь с ранее созданным (загруженным) битовым массивом (*Attach*). Кроме этого, класс обеспечивает набор функций, позволяющих менять и узнавать содержимое (*SetBitmapBits* и *GetBitmapBits*) и размеры (*SetBitmapDimension* и *GetBitmapDimension*) массива.

Сначала рассмотрим функции загрузки битового массива, созданного в качестве ресурса, а затем перейдем к функциям создания массивов непосредственно в памяти:

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

или

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

позволяют загрузить битовые массивы, заданные текстовой строкой (параметр lpszResourceName), т. е. по имени, или числовым идентификатором (параметр nIDResource). Обе эти функции позволяют загружать битовые массивы из ресурсов исполняемого модуля текущего приложения.[1]

Для загрузки системных битовых массивов используется функция *LoadOEMBitmap*:

```
BOOL CBitmap::LoadOEMBitmap(
  UINT nIDBitmap) —
```

в качестве идентификатора ресурса она может принимать одно из следующих значений, определенных в файле WINUSER.H:

OBM_LFARROW	OBM_LFARROWI	OBM_LFARROWD	OBM_OLD_LFARROW
OBM_RGARROW	OBM_RGARROWI	OBM_RGARROWD	OBM_OLD_RGARROW
OBM_DNARROW	OBM_DNARROWI	OBM_DNARROWD	OBM_OLD_DNARROW
OBM_UPARROW	OBM_UPARROWI	OBM_UPARROWD	OBM_OLD_UPARROW
OBM_COMBO			
OBM_MNARROW			
OBM_RESTORED	OBM_RESTORE		OBM_OLD_RESTORE
OBM_ZOOMD	OBM_ZOOM		OBM_OLD_ZOOM
OBM_REDUCED	OBM_REDUCE		OBM_OLD_REDUCE
OBM_CLOSE	OBM_OLD_CLOSE		
OBM_BTNCORNERS			
OBM_CHECKBOXES			
OBM_CHECK			
OBM_BTSIZE	OBM_SIZE		

[1] Вообще, загрузка ресурсов осуществляется при явном указании дескриптора программного модуля. В данном случае реализация функции будет использовать дескриптор приложения.

Идентификаторы, начинающиеся с OBM_OLD, связаны с пустыми битовыми массивами. Для того чтобы получить доступ к вышеперечисленным идентификаторам, в файл STDAFX.H необходимо включить строку

```
#define OEMRESOURCE
```

Далее в примере приведена функция, отображающая все системные битовые массивы (кроме "старых"). Следует иметь в виду, что размеры всех массивов определяются текущими установками и получение их размеров невозможно при помощи функции *GetBitmapDimension*, т. к. последняя возвращает размеры, только если ранее они были установлены при помощи функции *SetBitmapDimension*.

```
struct { int m_nBitmap; LPTSTR m_csBitmap; } btBitmap[] = {
  OBM_LFARROWI,    _T("OBM_LFARROWI"),
  OBM_LFARROWD,    _T("OBM_LFARROWD"),
  OBM_LFARROW,     _T("OBM_LFARROW"),

  OBM_RGARROWI,    _T("OBM_RGARROWI"),
  OBM_RGARROWD,    _T("OBM_RGARROWD"),
  OBM_RGARROW,     _T("OBM_RGARROW"),

  OBM_DNARROWI,    _T("OBM_DNARROWI"),
  OBM_DNARROWD,    _T("OBM_DNARROWD"),
  OBM_DNARROW,     _T("OBM_DNARROW"),

  OBM_UPARROWI,    _T("OBM_UPARROWI"),
  OBM_UPARROWD,    _T("OBM_UPARROWD"),
  OBM_UPARROW,     _T("OBM_UPARROW"),

  OBM_COMBO,       _T("OBM_COMBO"),
  OBM_MNARROW,     _T("OBM_MNARROW"),

  OBM_RESTORED,    _T("OBM_RESTORED"),
  OBM_RESTORE,     _T("OBM_RESTORE"),

  OBM_ZOOMD,       _T("OBM_ZOOMD"),
  OBM_ZOOM,        _T("OBM_ZOOM"),

  OBM_REDUCED,     _T("OBM_REDUCED"),
  OBM_REDUCE,      _T("OBM_REDUCE"),

  OBM_BTNCORNERS,  _T("OBM_BTNCORNERS"),

  OBM_CHECK,       _T("OBM_CHECK"),

  OBM_BTSIZE,      _T("OBM_BTSIZE"),
  OBM_SIZE,        _T("OBM_SIZE"),

  OBM_CHECKBOXES,  _T("OBM_CHECKBOXES"),

  OBM_CLOSE,       _T("OBM_CLOSE") };

...
```

```
void COEMBitmapFrame::OnPaint()
{
  CPaintDC dc(this);

  TEXTMETRIC tm;
  dc.GetTextMetrics(&tm);

  // устанавливаем режим "прозрачного" вывода
  dc.SetBkMode(TRANSPARENT);

  int nY = 10;
  int nX = 10;

  CBitmap bitmap;

  for (int i = 0; i < sizeof(btBitmap) / sizeof(btBitmap[0]); i++)
  {
    // загружаем системный битовый массив
    bitmap.LoadOEMBitmap(btBitmap[i].m_nBitmap);
    //
    // получаем размеры, заполняя структуру
    // BITMAP, которая определена в Win32 API
    // для описания битовых массивов
    //
    BITMAP bmp;

    // первый способ - используем функцию GetObject
    bitmap.GetObject(sizeof(BITMAP), &bmp);

    // второй способ - используем функцию GetBitmap
    // bitmap.GetBitmap(&bmp);

    CSize sz(bmp.bmWidth, bmp.bmHeight);

    dc.DrawState(CPoint(nX, nY), sz,
      bitmap, DST_BITMAP);

    dc.TextOut(nX+60,
      nY-tm.tmInternalLeading, btBitmap[i].m_csBitmap);

    // сдвиг вниз на 5 единиц, в данном случае
    // пикселей, больше, чем максимальная высота
    // битового массива или соответствующего текста
    nY += max(sz.cy, tm.tmHeight) + 5;

    bitmap.DeleteObject();

    // выводим изображения и текст в две колонки
    if (i == sizeof(btBitmap) / sizeof(btBitmap[0]) / 2)
    {
      nX += 200;
      nY = 10;
    }
  }
}
```

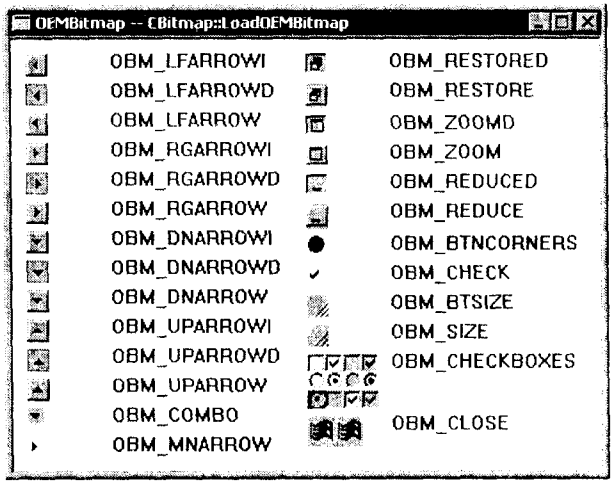


Рис. 8. Результат работы функции COEMBitmapFrame::OnPaint

Обратите внимание, что именно эти "значки" используются в Windows для рисования многих элементов управления. Причем при смене параметров функционирования графической системы меняются и эти элементы управления (их размеры и цвет), так что приложения, пользующиеся системными битовыми массивами, могут так же легко приспосабливаться к изменившимся условиям!

Далее мы рассмотрим два примера, задача которых создать битовые массивы прямо в памяти, путем "создания" картинки в массивах данных. Но перед этим рассмотрим функции-члены класса *CBitmap*, которые используются в примерах.

```
BOOL CBitmap::CreateCompatibleBitmap(
   CDC* pDC,
   int nWidth,
   int nHeight)
```

Функция предназначена для создания битовых массивов, совместимых с конкретным устройством вывода. Windows поддерживает два вида растровых изображений: независимых от устройства (Device Independent Bitmap) и, соответственно, зависимых. Данная функция создает битовые массивы последнего типа, но т. к. изображение готовится к выводу на конкретном устройстве — это вполне нормально. В качестве параметров функция принимает указатель на объект класса *CDC* (параметр *pDC*), для которого и создается изображение, параметры *nWidth* и *nHeight* определяют ширину и высоту соответственно. После выполнения этой функции битовый массив заполнен случайными значениями цветов для каждой точки. Такие битовые массивы могут быть либо заполнены необходимыми значениями для каждой точки, либо они могут являться основой контекстов устройств, расположенных в памяти. Для такого контекста выполнимы все функции рисования.

Другая функция создает битовый массив для вполне конкретного устройства:

```
BOOL CBitmap::CreateBitmap(
  int nWidth,
  int nHeight,
  UINT nPlanes,
  UINT nBitcount,
  const void* lpBits)
```

Параметры *nWidth* и *nHeight*, как и для предыдущей функции, определяют размеры изображения. Параметры *nPlanes* и *nBitCount* определяют параметры хранения информации о цвете: количество слоев и количество битов для хранения одного цвета. Наконец, параметр *lpBits* является указателем на массив, в котором хранятся значения цветов для каждой точки. К сожалению, формат хранения такой информации строго зависит от устройства вывода.

В функции CCreateBmpOne::OnPaint мы реализовали два способа создания битового массива. Первый состоит в том, что после создания совместимого с контекстом устройства окна битового массива мы заполнили его ранее подготовленными значениями цветов (функция *SetBitmapBits*). Второй способ заключается в создании и одновременной инициализации битового массива. Недостаток этих подходов в том, что в обоих случаях предполагается, что мы имеем дело с устройством, требующим для хранения цвета одного пикселя 16 бит, причем 5 бит отводится под красную составляющую, 6 бит — под зеленую и 5 бит — под синюю. Созданный таким образом битовый массив не будет верно отображаться на устройствах с другой организацией.

```
void CCreateBmpOneFrame::OnPaint()
{
  CPaintDC dc(this);

  int CX = 400, CY = 100;

  // первое допущение: надеемся, что количество бит,
  // необходимых для данных типа WORD, — 16
  WORD *lpvBits = new WORD[CX * CY];

  for (int i = 0; i < CY; i++)
  {
    for (int j = 0; j < CX; j++)
    {
      WORD color = 31-abs(i*31/(CY/2)-31);

      // второе допущение: надеемся, что устройство вывода
      // требует для хранения цвета 16 бит, причем
      // для красного — старшие 5
      // для зеленого — средние 6
      // для синего   — младшие 5
```

```
      lpvBits[i*CX+j] =
        (WORD)((color << 11) + (color*2 << 5) + color);
    }
  }

  CBitmap bitmap;

  // первый вариант создания битового массива
  bitmap.CreateCompatibleBitmap(&dc, CX, CY);
  bitmap.SetBitmapBits(sizeof(WORD) * CX * CY, lpvBits);

  // второй вариант создания битового массива
  // bitmap.CreateBitmap(CX, CY, 1, 16, lpvBits);

  // будем надеяться, что что-нибудь будет выведено
  dc.DrawState(CPoint(10, 10),
    CSize(CX, CY), &bitmap, DST_BITMAP);

  delete lpvBits;
}
```

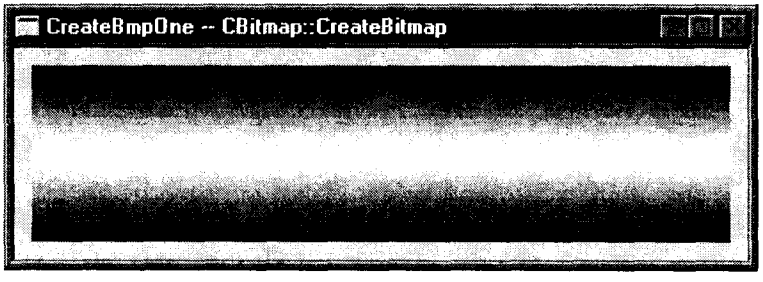


Рис. 9. Результат работы функции CCreateBmpOneFrame::OnPaint — система требует 16 битов на цвет

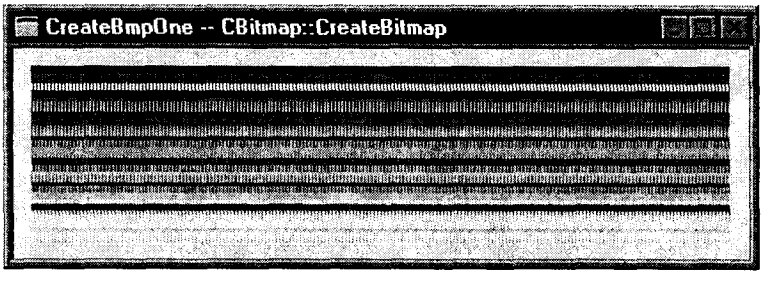


Рис. 10. Результат работы функции CCreateBmpOneFrame::OnPaint — система требует 8 битов на цвет

В функции CCreateBmpTwo::OnPaint реализован второй способ создания битового массива. В примере используется функция Win32 API *CreateDIBitmap*, которая принимает, в качестве информации о цвете точки,

RGB-тройки, что существенно упрощает программирование и делает код независимым от конкретных настроек видеосистемы.

```
void CCreateBmpTwo::OnPaint()
{
  CPaintDC dc(this);

  int CX = 400, CY = 100;

  DWORD *lpbInit = new DWORD[CX * CY];

  for (int i = 0; i < CY; i++)
  {
    for (int j = 0; j < CX; j++)
    {
      BYTE color = (BYTE)(255-abs(i*255/(CY/2)-255));

      lpbInit[i*CX+j] = RGB(color, color, color);
    }
  }

  BITMAPINFO bmi = {
    sizeof(BITMAPINFOHEADER), CX, CY, 1, 32, BI_RGB, 0,
    0, 0, 0, 0
  };

  HBITMAP hBitmap = ::CreateDIBitmap(
    dc.GetSafeHdc(), &bmi.bmiHeader, CBM_INIT,
    lpbInit, &bmi, DIB_RGB_COLORS);

  CBitmap bitmap;
  bitmap.Attach(hBitmap);

  dc.DrawState(CPoint(10, 10), CSize(CX, CY),
    &bitmap, DST_BITMAP);

  bitmap.DeleteObject();

  delete lpbInit;
}
```

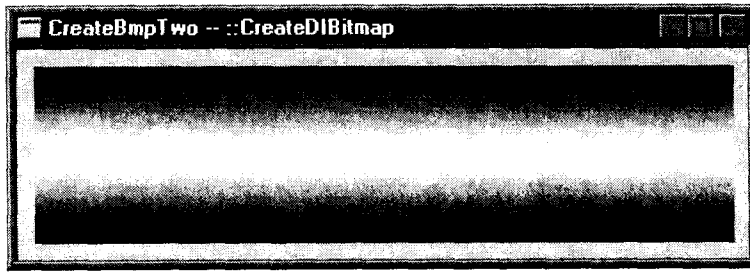


Рис. 11. Результат работы функции CCreateBmpTwoFrame::OnPaint — система требует 16 битов на цвет

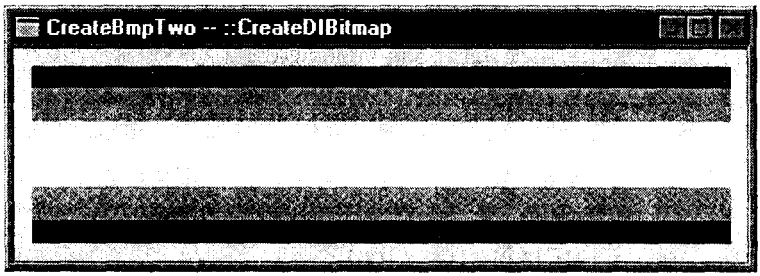
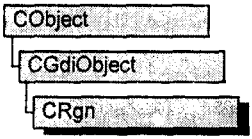


Рис. 12. Результат работы функции CCreateBmpTwoFrame::OnPaint — система требует 8 битов на цвет

Регион (класс CRgn)



Класс *CRgn* инкапсулирует объект Windows "регион", который может представлять собой совокупность прямоугольных и эллиптических областей. Основное назначение регионов — определение области отсечения в пределах окна. Класс *CRgn* имеет в своем составе набор функций для создания регионов, схожих по своим параметрам с функциями рисования прямоугольников, эллипсов и многоугольников, а также функции, позволяющие комбинировать регионы.

Создание регионов

Для создания региона при помощи класса *CRgn* необходимо использовать конструктор без параметров, который создает объект класса *CRgn*, но не создает при этом собственно объект Windows "регион". Для создания последнего можно воспользоваться функциями, описанными ниже.

Создание прямоугольных регионов

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

и

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

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

Создание эллиптических регионов

```
BOOL CRgn::CreateEllipticRgn(
   int x1,
   int y1,
   int x2,
   int y2)
и
BOOL CRgn::CreateEllipticRgnIndirect(
   LPCRECT lpRect)
```

Обе функции получают информацию, аналогично функциям предыдущей группы, о прямоугольной области, в которую вписан эллипс.

Создание сложных регионов на базе многоугольников

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

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

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

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

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

Создание регионов с закругленными углами

```
BOOL CRgn::CreateRoundRectRgn(
   int x1,
   int y1,
   int x2,
   int y2,
   int x3,
   int y3)
```

Первые четыре параметра определяют прямоугольную область, а параметры *x3* и *y3* определяют ширину и высоту эллипса, на основе которого строятся закругленные углы.

Комбинирование регионов

Более подробно остановимся на функции, которая на основе комбинации двух регионов изменяет третий. Именно *изменяет*, т. к. объект класса, для которого вызывается эта функция и который собственно и будет хранить результат комбинации двух других регионов, обязательно должен существовать[1] — так работает Windows, так работает и библиотека классов MFC.

```
int CRgn::CombineRgn(
  CRgn* pRgn1,
  CRgn* pRgn2,
  int nCombineMode)
```

Функция возвращает одно из следующих значений, показывающих, какого типа регион получился в результате комбинирования:

COMPLEXREGION	регион состоит из набора более простых областей
ERROR	произошла ошибка
NULLREGION	регион пуст
SIMPLEREGION	регион представляет собой одну простую область — прямоугольник или эллипс

Параметры *pRgn1* и *pRgn2* являются указателями на объекты класса *CRgn*, которые должны быть связаны с регионами Windows, именно они и будут комбинироваться. Параметр *nCombineMode* определяет режим комбинирования. Таких режимов несколько, мы уже упоминали их в предыдущей главе, но в несколько другом контексте, поэтому перечислим их еще раз:

RGN_AND	регион будет являться пересечением обоих заданных, т. е. определяться областью, которая относится и к первому и ко второму региону одновременно
RGN_COPY	регион будет точной копией первого региона (pRgn1)
RGN_DIFF	регион будет включать в себя только те области первого региона (pRgn1), которые не пересекаются со вторым (pRgn2)
RGN_OR	регион будет являться объединением обоих заданных, т. е. будет включать все области, относящиеся и к первому и ко второму региону
RGN_XOR	регион будет включать области первого региона (pRgn1) и второго региона (pRgn2) кроме тех, которые принадлежат и первому и второму одновременно

Несмотря на то, что Windows требует наличия объектов для всех трех регионов для осуществления комбинирования, возможно использование в качест-

[1] Объект класса *CRgn*, для которого вызывается функция, может совпадать с одним из тех, указатели на которые передаются в качестве параметров.

ве одного или даже обоих параметров указателя на объект класса *CRgn*, для которого и вызывается функция *CombineRgn*:

```
rgn.CombineRgn(&rgn, &rgnPrev);
```

Кроме вышеперечисленных функций-членов упомянем еще несколько. Класс *CRgn* имеет в своем составе функцию копирования *CopyRgn*, функцию преобразования текущего контура (path) в регион (*CreateFromPath*), функцию сравнения двух регионов (*EqualRgn*).

В качестве примера использования регионов приведем функцию, в которой, в результате комбинирования трех регионов, производится закрашивание части замкнутых областей одним вызовом функции рисования.

```
void CRegionFrame::OnPaint()
{
  CPaintDC dc(this);

  CRect rect;
  GetClientRect(rect);

  // определяем треугольную область с одной вершиной
  // в центре клиентской части окна и двумя другими,
  // совпадающими с левым верхним и левым нижним углами
  POINT pt_1[4] = { 0, 0,
                    rect.right / 2, rect.bottom / 2,
                    0, rect.bottom,
                    0, 0 };

  CRgn rgn_1;
  rgn_1.CreatePolygonRgn(pt_1, 4, WINDING);

  // определяем треугольную область с одной вершиной
  // в центре клиентской части окна и двумя другими,
  // совпадающими с правым верхним и правым нижним углами
  POINT pt_2[4] = { rect.right, 0,
                    rect.right / 2, rect.bottom / 2,
                    rect.right, rect.bottom,
                    rect.right, 0 };

  CRgn rgn_2;
  rgn_2.CreatePolygonRgn(pt_2, 4, WINDING);

  // определяем прямоугольную область, лежащую точно в
  // центре клиентской области окна и немного меньше ее
  CRgn rgn;
  rgn.CreateRectRgn(40, 40, rect.right-40, rect.bottom-40);

  // определяем регион как комбинацию трех заданных
  rgn.CombineRgn(&rgn, &rgn_1, RGN_XOR);
  rgn.CombineRgn(&rgn, &rgn_2, RGN_XOR);
```

```
    //
    // определяем параметры рисования
    //

    // кисть
    CBrush brush(HS_DIAGCROSS, RGB(0, 0, 0));
    CBrush *pBrush = dc.SelectObject(&brush);

    // цвет фона
    dc.SetBkColor(RGB(128, 128, 128));

    // область отсечения
    dc.SelectObject(&rgn);

    // рисуем прямоугольник на всю клиентскую часть окна
    dc.Rectangle(rect);

    dc.SelectObject(pBrush);
}
```

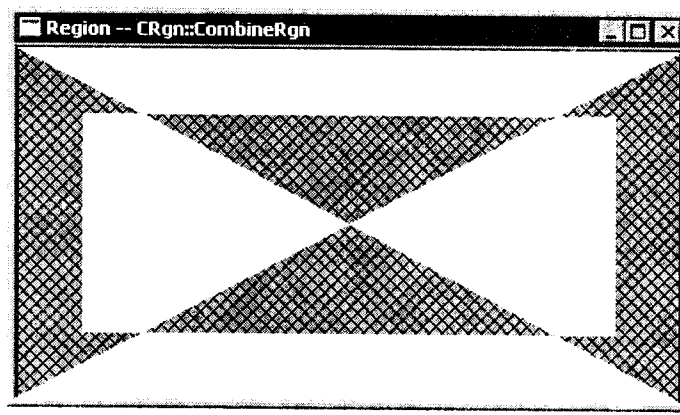


Рис. 13. Результат работы функции CRegionFrame::OnPaint

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

Глава 19

Пример приложения PQClock

Эта глава полностью посвящена исходным текстам приложения, задача которого отображать текущее время в виде аналоговых часов (т. е. часов с циферблатом и со стрелками). Естественно, что основная цель нами преследовалась несколько иная, а именно — показать использование большей части того, чему были посвящены три предыдущие главы.

Далее приводится лишь основная часть[1] исходных текстов приложения-примера. Это два файла (заголовочный и файл кода) класса приложения *CPQClockApp* и два файла (заголовочный и файл кода) класса главного окна приложения *CClockFrame*. Большое количество комментариев позволит лучше и быстрее понять суть примера, используемые классы и алгоритмы.

Класс приложения

```
/*************************************************************
Файл PQClock.h
Copyright (c) 1997 Мешков А., Тихомиров Ю., Шведов Д.
*************************************************************/

// PQClock.h : main header file for the PQClock application
//

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

#include "resource.h"       // main symbols

////////////////////////////////////////////////////////////////
// CPQClockApp:
// See PQClock.cpp for the implementation of this class
//

class CPQClockApp : public CWinApp
```

[1] Полностью исходные тексты приложения PQClock и его исполняемый модуль можно найти на прилагаемой к этому тому дискете.

```
{
public:
  CPQClockApp();

// Overrides
  // ClassWizard generated virtual function overrides
  //{{AFX_VIRTUAL(CPQClockApp)
  public:
  virtual BOOL InitInstance();
  virtual BOOL OnIdle(LONG lCount);
  //}}AFX_VIRTUAL

// Implementation

  //{{AFX_MSG(CPQClockApp)
    // NOTE - the ClassWizard will add and
    // remove member functions here.
    //    DO NOT EDIT what you see in these
    //    blocks of generated code !
  //}}AFX_MSG
  DECLARE_MESSAGE_MAP()
};

/////////////////////////////////////////////////////////////////////////////

/*************************************************************
Файл PQClock.cpp
Copyright (c) 1997 Мешков А., Тихомиров Ю., Шведов Д.
*************************************************************/

// PQClock.cpp : Defines the class behaviors for the application.
//

#include "stdafx.h"
#include "PQClock.h"
#include "ClockFrm.h"

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

/////////////////////////////////////////////////////////////////////////////
// CPQClockApp

BEGIN_MESSAGE_MAP(CPQClockApp, CWinApp)
  //{{AFX_MSG_MAP(CPQClockApp)
    // NOTE - the ClassWizard will add and
    // remove mapping macros here.
    //    DO NOT EDIT what you see in these
    //    blocks of generated code!
```

```
  //}}AFX_MSG
  ON_COMMAND(ID_HELP, CWinApp::OnHelp)
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CPQClockApp construction

CPQClockApp::CPQClockApp()
{
  // TODO: add construction code here,
  // Place all significant initialization in InitInstance
}

/////////////////////////////////////////////////////////////////////////////
// The one and only CPQClockApp object

CPQClockApp theApp;

/////////////////////////////////////////////////////////////////////////////
// CPQClockApp initialization

BOOL CPQClockApp::InitInstance()
{
  // Standard initialization
  // If you are not using these features and
  // wish to reduce the size
  // of your final executable,
  // you should remove from the following
  // the specific initialization routines you do not need.

#ifdef _AFXDLL
  Enable3dControls();     // Call this when using MFC in a shared DLL
#else
  Enable3dControlsStatic(); // Call this when linking to MFC
                            // statically
#endif

  CClockFrame *pWnd = new CClockFrame;
  m_pMainWnd = pWnd;

  int nSize = min(GetSystemMetrics(SM_CXMAXIMIZED),
    GetSystemMetrics(SM_CYMAXIMIZED)) * 3 / 5;

  pWnd->Create(NULL, "Часы 'PRAGMA Quartz'",
    WS_OVERLAPPEDWINDOW, CRect(0, 0, nSize, nSize));

  pWnd->CenterWindow();
  pWnd->ShowWindow(m_nCmdShow);
  pWnd->UpdateWindow();

  return TRUE;
}
```

```
BOOL CPQClockApp::OnIdle(LONG lCount)
{
  CWinApp::OnIdle(lCount);

  // перерисовываем окно
  ((CClockFrame *)m_pMainWnd)->Redraw();

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

Обратите внимание, что именно в функции *OnIdle* происходит вызов процедуры перерисования часов — при этом не задействуется таймер или не добавляется код в функцию *WinMain* (которой у нас и нет), как это необходимо было бы сделать в рамках Win32 API.

Вот как выглядит окно приложения, функции собственно отображения которого приведены далее:

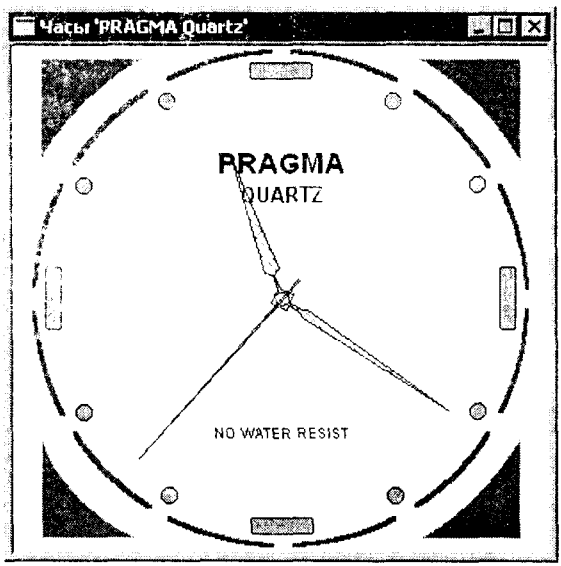


Рис. 14. Окно приложения PQClock при начальных размерах окна

Класс окна

```
/*************************************************************
Файл ClockFrm.h
Copyright (c) 1997 Мешков А., Тихомиров Ю., Шведов Д.
*************************************************************/

// ClockFrm.h : header file
//
```

```
/////////////////////////////////////////////////////////////////////////
// CClockFrame frame

class CClockFrame : public CFrameWnd
{
public:
  CClockFrame();

// Attributes
public:

// Operations
public:
  void Redraw();

private:
  void RotatePoint(POINT pt[], int nNum, double nAngle);
  void SetMode(CDC *pDC);
  void DrawClock(CDC *pDC);
  void DrawLabel(CDC *pDC);
  void DrawArrows(CDC *pDC, CTime &tm);

// Overrides
  // ClassWizard generated virtual function overrides
  //{{AFX_VIRTUAL(CClockFrame)
  //}}AFX_VIRTUAL

// Implementation
protected:
  virtual ~CClockFrame();

  // Generated message map functions
  //{{AFX_MSG(CClockFrame)
  afx_msg void OnSize(UINT nType, int cx, int cy);
  afx_msg void OnPaint();
  afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
  //}}AFX_MSG
  DECLARE_MESSAGE_MAP()

private:
  int m_cxClient;
  int m_cyClient;

  CFont m_Font_1;
  CFont m_Font_2;
  CFont m_Font_3;

  CBrush m_wndBrush;
  CPen m_wndPen;

  CBrush m_arrBrush;
  CPen m_arrPen;
```

```
  CBrush m_clkBrush;
  CPen m_clkPen;
  CBrush m_rndBrush;
  CPen m_rndPen;
  CBrush m_crnBrush;
  CPen m_crnPen;
};

/////////////////////////////////////////////////////////////////////////

/*************************************************************
Файл ClockFrm.cpp
Copyright (c) 1997 Мешков А., Тихомиров Ю., Шведов Д.
*************************************************************/

// ClockFrm.cpp : implementation file
//

#include "stdafx.h"
#include "PQClock.h"
#include "ClockFrm.h"

#include "math.h"

// для возможности быстрого изменения цветов
// определим их в одном месте -
// в начале файла
#define COLOR_BCKGR   RGB(224,224,224)
#define COLOR_WHITE   RGB(255,255,255)
#define COLOR_BLACK   RGB(  0,  0,  0)
#define COLOR_ARROW   RGB(164,164,  0)
#define COLOR_CLOCK   RGB(164,164,  0)
#define COLOR_ROUND   RGB( 92, 92,  0)
#define COLOR_CRNER   RGB( 92, 92,  0)

#define TWOPI         (2 * 3.14159)

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

/////////////////////////////////////////////////////////////////////////
// CClockFrame

CClockFrame::CClockFrame()
{
}

CClockFrame::~CClockFrame()
{
}
```

```
//
// функция вычисления текущих координат точек
// в зависимости от угла поворота
//
void CClockFrame::RotatePoint(POINT pt[], int nNum, double nAngle)
{
  POINT ptTemp;

  // поворот осуществляется вокруг начала координат
  for (int i = 0; i < nNum; i++)
  {
    ptTemp.x = (int)(pt[i].x * cos (TWOPI * nAngle / 360) +
                     pt[i].y * sin (TWOPI * nAngle / 360));

    ptTemp.y = (int)(pt[i].y * cos (TWOPI * nAngle / 360) -
                     pt[i].x * sin (TWOPI * nAngle / 360));
    pt[i] = ptTemp;
  }
}

//
// функция формирования системы координат
//
void CClockFrame::SetMode(CDC *pDC)
{
  // устанавливаем масштаб, начало и направление
  // координатных осей
  pDC->SetMapMode(MM_ISOTROPIC);
  pDC->SetWindowExt(1000, 1000);
  pDC->SetViewportExt(m_cxClient / 2, -m_cyClient / 2);
  pDC->SetViewportOrg(m_cxClient / 2,  m_cyClient / 2);

  // в результате:
  //   начало координат в середине клиентской области окна
  //   ось "x" направлена влево
  //   ось "y" направлена вверх
  //   размер окна 2000 X 2000 логических единиц
  //     X: [-1000; +1000]
  //     Y: [-1000; +1000]
}

//
// функция рисования циферблата часов
//
void CClockFrame::DrawClock(CDC *pDC)
{
  CPen *pPen;
  CBrush *pBrush;

  int nAngle;
```

```
//
// рисуем отметки для каждого часа,
// причем для 12, 3, 6 и 9 часов
// отметки - прямоугольники,
// для остальных - круги
//

pBrush = pDC->SelectObject(&m_clkBrush);

for (nAngle = 0; nAngle < 360; nAngle += 30)
{
  POINT pt[2] = { 0, 900 };

  RotatePoint(pt, 1, nAngle);

  int x = (nAngle % 180) ? 70 : 250;
  int y = ((nAngle+90) % 180) ? 70 : 250;

  pt[0].x -= x / 2;
  pt[0].y -= y / 2;

  pt[1].x  = pt[0].x + x;
  pt[1].y  = pt[0].y + y;

  if (nAngle % 90)
    pDC->Ellipse(pt[0].x, pt[0].y, pt[1].x, pt[1].y);
  else
  {
    pDC->RoundRect(pt[0].x, pt[0].y, pt[1].x, pt[1].y,
      min(x, y) / 2, min(x, y) / 2);
  }
}

//
// рисуем дуги между часовыми отметками
// так, чтобы их диаметр был немного
// больше окружности, на которой расположены
// часовые отметки
//

pPen = pDC->SelectObject(&m_rndPen);

POINT ptPrev;

for (nAngle = -30; nAngle < 360; nAngle += 30)
{
  POINT pt[2] = { -45, 950, 45, 950 };

  RotatePoint(pt, 2, nAngle);

  if (nAngle != -30)
  {
    pDC->Arc(-980, -980, 980, 980,
```

```
        pt[0].x, pt[0].y, ptPrev.x, ptPrev.y);
    }

    ptPrev = pt[1];
}

//
// рисуем "углы" часов, причем
// четырьмя различными способами
//

CRect round(-1100, -1100, 1100, 1100);

CRgn rgn;
CRect rect;

pDC->SelectObject(&m_crnPen);
pDC->SelectObject(&m_crnBrush);


//
// 1-ый способ:
// рисуем левый верхний угол
//

rect = CRect(-950, 950, -550, 550);

//
// формируем область отсечения
// для этого нужны координаты в единицах
// устройства вывода
pDC->LPtoDP(&rect);

rgn.CreateRectRgnIndirect(&rect);
pDC->SelectObject(&rgn);

// рисуем замкнутую дугу (круг), но выводится
// только ее часть в соответствии с заданной
// областью отсечения
pDC->Arc(&round, CPoint(0, 0), CPoint(0, 0));

// добавляем две линии для формирования
// замкнутого контура
pDC->MoveTo(-550, 950);
pDC->LineTo(-950, 950);
pDC->LineTo(-950, 550);

// разрешаем для рисования всю клиентскую часть окна
pDC->SelectClipRgn(NULL);

// заполняем замкнутую область
// от внутренней точки до границы
pDC->ExtFloodFill(-900, 900, COLOR_CRNER, FLOODFILLBORDER);
```

```
//
// 2-ой способ:
// рисуем правый верхний угол
//

// рисуем дугу и две линии для формирования
// замкнутого контура
pDC->Arc(&round, CPoint(950, 550), CPoint(550, 950));
pDC->MoveTo(550, 950);
pDC->LineTo(950, 950);
pDC->LineTo(950, 550);

// заполняем замкнутую область, заданную цветом,
// с внутренней точки
pDC->ExtFloodFill(900, 900, COLOR_BCKGR, FLOODFILLSURFACE);

//
// 3-ий способ:
// рисуем правый нижний угол
//

//
// создаем два региона
//

// первый - прямоугольный, в него
//   вписана окружность
rect = CRect(550, -950, 950, -550);

CRgn rgn1;
rgn1.CreateRectRgnIndirect(&rect);

// второй - эллиптический, он как раз
//   равен окружности
CRect rect2 = round;

CRgn rgn2;
rgn2.CreateEllipticRgnIndirect(&rect2);

// на основе созданных регионов создаем
// комбинированный с нужной нам формой
rgn.CombineRgn(&rgn1, &rgn2, RGN_DIFF);

// закрашиваем его
pDC->PaintRgn(&rgn);

// в явном виде объекты можно не удалять -
// деструктор сделает все за нас !
// rgn.DeleteObject();
// rgn1.DeleteObject();
// rgn2.DeleteObject();
```

```
  //
  // 4-ый способ:
  // рисуем левый нижний угол
  //

  //
  // начинаем формирование контура
  //

  pDC->BeginPath();

  // массив точек для рисования
  // кривой Безье
  POINT pt[] = {
    -950, -550,
    -950, -670,
    -670, -950,
    -550, -950
  };

  // "рисуем" кривую Безье, т. к. другие
  // функции рисования гладких кривых
  // не фиксируются в контуре
  pDC->PolyBezier(pt, 4);

  // добавляем две линии для формирования
  // замкнутого контура
  pDC->MoveTo(-950, -550);
  pDC->LineTo(-950, -950);
  pDC->LineTo(-550, -950);

  pDC->EndPath();

  // закрашиваем пространство, ограниченное
  // сформированным контуром
  pDC->FillPath();

  pDC->SelectObject(pPen);
  pDC->SelectObject(pBrush);
}

//
// функция рисования строк текста на циферблате
//
void CClockFrame::DrawLabel(CDC *pDC)
{
  pDC->SetTextAlign(TA_CENTER);

  // первые две строки рисуем так, чтобы
  // промежутки заполнялись цветом фона
  COLORREF color = pDC->SetBkColor(COLOR_BCKGR);
```

```
  CFont *pFont = pDC->SelectObject(&m_Font_1);
  pDC->TextOut(0,  600, _T("PRAGMA"), 6);

  TEXTMETRIC tm;
  pDC->GetTextMetrics(&tm);

  pDC->SelectObject(&m_Font_2);
  pDC->TextOut(0,  600-(tm.tmHeight+tm.tmExternalLeading),
    _T("QUARTZ"), 6);

  //
  // последнюю строку рисуем так, чтобы
  // промежутки между контурами символов
  // оставались без изменения
  //

  // цвет фона не оказывает влияния на эти области
  pDC->SetBkColor(color);

  pDC->SetBkMode(TRANSPARENT);

  pDC->SelectObject(&m_Font_3);
  pDC->TextOut(0, -500, _T("NO WATER RESIST"));

  pDC->SelectObject(pFont);
}

//
// функция вывода стрелок установленными вне
// нее текущими карандашом и кистью
//
void CClockFrame::DrawArrows(CDC *pDC, CTime &tm)
{
  // массив точек, которые определяют контуры стрелок
  static POINT pt[3][11] =
  {
      0,  -50, -50,    0, -5,   40, -5,  100, -30,  120,
      0,  600,  30,
    120,    5, 100,    5, 40,   50,  0,    0, -50,

      0,  -40, -40,    0, -5,   30, -5,   90, -20,  110,
      0,  800,
     20,  110,   5,   90,  5,   30, 40,    0,   0,  -40,

      0, -100,  -5, -100, -5, -100, -5, -100,  -5, -100,
      0,  850,
      5, -100,   5, -100,  5, -100,  5, -100,   0, -100
  };

  double nAngle[3];

  // вычисление углов поворота каждой из трех стрелок
  nAngle[0] = (tm.GetHour()    * 30) % 360 + tm.GetMinute() / 2;
```

```
  nAngle[1] = (tm.GetMinute() * 60 + tm.GetSecond()) / 10.0;
  nAngle[2] = (tm.GetSecond() *  6);

  POINT ptTemp[3][11];
  memcpy(ptTemp, pt, sizeof(pt));

  for (int i = 0; i < 3; i++)
  {
    // вычисление текущих координат всех точек
    // контура стрелки
    RotatePoint(ptTemp[i], 11, nAngle[i]);

    // рисование стрелки
    pDC->Polygon(ptTemp[i], 11);
  }
}

//
// функция перерисовки часов, может вызываться
// практически в любой функции программы
//
void CClockFrame::Redraw()
{
  static CTime m_tmPrev;

  if (IsIconic())
    return;

  CClientDC dc(this);

  CTime tmCurrent = CTime::GetCurrentTime();

  if (tmCurrent != m_tmPrev)
  {
    // устанавливаем нашу систему координат
    SetMode(&dc);

    // рисуем циферблат
    DrawClock(&dc);

    // закрашиваем цветом фона ранее выведенные стрелки
    CPen *pPen = dc.SelectObject(&m_wndPen);
    CBrush *pBrush = dc.SelectObject(&m_wndBrush);

    DrawArrows(&dc, m_tmPrev);

    // выводим текст надписей
    DrawLabel(&dc);

    // выводим стрелки по их новому месту
    dc.SelectObject(&m_arrPen);
    dc.SelectObject(&m_arrBrush);

    DrawArrows(&dc, tmCurrent);
```

```
    dc.SelectObject(pBrush);
    dc.SelectObject(pPen);
  }

  m_tmPrev = tmCurrent;
}

BEGIN_MESSAGE_MAP(CClockFrame, CFrameWnd)
  //{{AFX_MSG_MAP(CClockFrame)
  ON_WM_SIZE()
  ON_WM_PAINT()
  ON_WM_CREATE()
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()

////////////////////////////////////////////////////////////////////////
// CClockFrame message handlers

void CClockFrame::OnSize(UINT nType, int cx, int cy)
{
  CFrameWnd::OnSize(nType, cx, cy);

  // запоминаем размер клиентской области
  // окна, чтобы каждый раз при перерисовке
  // не запрашивать размер заново
  m_cxClient = cx;
  m_cyClient = cy;
}

void CClockFrame::OnPaint()
{
  CPaintDC dc(this);

  // закрашиваем клиентскую область
  CBrush brush(COLOR_BCKGR);
  dc.FillRect(
    CRect(0, 0, m_cxClient, m_cyClient), &brush);

  // циферблат, текст и стрелки рисовать не будем,
  // этим займется функция CPQClock::OnIdle
}

int CClockFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
  if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
    return -1;

  //
  // для экономии времени вместо того, чтобы при
  // рисовании создавать шрифты, карандаши и кисти,
  // сделаем это один раз.
  // При этом надо помнить, что захваченные таким
```

```
// образом ресурсы будут освобождены при удалении
// объекта класса CClockFrame – объекты соответствующих
// классов являются членами класса CClockFrame
//

m_Font_1.CreateFont(130, 0, 0, 0, FW_BOLD,
  0, 0, 0, DEFAULT_CHARSET,
  OUT_DEFAULT_PRECIS, CLIP_DEFAULT_PRECIS, PROOF_QUALITY,
  VARIABLE_PITCH | FF_DONTCARE, "Arial");

m_Font_2.CreateFont(100, 0, 0, 0, FW_BOLD,
  0, 0, 0, DEFAULT_CHARSET,
  OUT_DEFAULT_PRECIS, CLIP_DEFAULT_PRECIS, PROOF_QUALITY,
  VARIABLE_PITCH | FF_SWISS, "Arial");

m_Font_3.CreateFont(70, 0, 0, 0, FW_BOLD,
  0, 0, 0, DEFAULT_CHARSET,
  OUT_DEFAULT_PRECIS, CLIP_DEFAULT_PRECIS, PROOF_QUALITY,
  VARIABLE_PITCH | FF_SWISS, "Arial");

m_wndBrush.CreateSolidBrush(COLOR_BCKGR);
m_wndPen.CreatePen(PS_SOLID, 1, COLOR_BCKGR);

m_arrBrush.CreateHatchBrush(HS_DIAGCROSS, COLOR_ARROW);
m_arrPen.CreatePen(PS_SOLID, 1, COLOR_BLACK);

m_clkBrush.CreateSolidBrush(COLOR_CLOCK);
m_clkPen.CreatePen(PS_SOLID, 1, COLOR_BLACK);

m_rndBrush.CreateSolidBrush(COLOR_ROUND);

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

LOGBRUSH lb;
m_rndBrush.GetLogBrush(&lb);
m_rndPen.CreatePen(PS_SOLID | PS_GEOMETRIC |
  PS_ENDCAP_SQUARE, 20, &lb);

m_crnBrush.CreateSolidBrush(COLOR_CRNER);
m_crnPen.CreatePen(PS_SOLID, 1, COLOR_CRNER);

 return 0;
}
```

Используемая методика рисования в режиме MM_ISOTROPIC позволяет не добавлять ни строчки кода для настройки или перенастройки параметров (прежде всего координат) при изменении размеров окна. Сравните следующий рисунок, с тем, что был приведен ранее — размеры другие, а содержимое окна то же.[1]

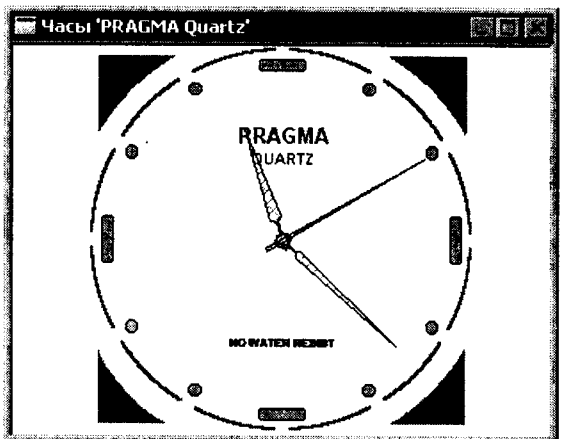


Рис. 15. Окно приложения PQClock после изменения размеров окна

Часть VIII

Стандартные элементы управления (Controls)

20. Обзор стандартных элементов управления

21. Классы стандартных элементов управления

22. Классы с расширенными возможностями

Глава 20

Обзор стандартных элементов управления

Библиотека MFC обеспечивает набор классов, которые облегчают создание стандартных элементов управления Windows. Они включают кнопки различных видов, статический текст, поле редактирования текста, полосы прокрутки, список и комбинированный список. В табл. 3 приведены названия классов и стандартные элементы управления, создаваемые этими классами.

Таблица 3. *Классы MFC для создания стандартных элементов управления*

Класс	Элемент управления
CStatic	статический текст или изображение
CButton	командные кнопки, срабатывающие по нажатию, переключатели и флажки
CListBox	список
CComboBox	комбинированный блок списка и поля редактирования
CEdit	поле редактирования
CScrollBar	полоса прокрутки

Кроме вышеперечисленных классов, библиотека обеспечивает разработчиков еще тремя классами, которые, базируясь на стандартных элементах управления и соответствующих классах, существенно упрощают создание и управление элементами управления с расширенными возможностями. Эти возможности ранее можно было реализовать только самостоятельно, написав обработчики для ряда сообщений. Теперь можно просто пользоваться тем, что сделано другими.

Создание элементов управления

Большинство элементов управления (стандартные — все) могут быть созданы двумя способами:

- в редакторе ресурсов
- прямо в исходном тексте приложения

Использование редактора ресурсов для создания элементов управления

Элементы управления можно создавать во время создания шаблона блока диалога при помощи редактора ресурсов. При этом пользователь может просто перетаскивать с помощью мыши необходимые элементы управления из панели с их изображениями в создаваемое диалоговое окно, изменяя по своему желанию размеры, расположение и параметры выбранного элемента управления. При этом редактор ресурсов помещает описания создаваемых элементов блока диалога в файл ресурсов. Когда диалоговое окно будет создаваться (вызовом функций *Create* или *DoModal*), Windows создаст элементы управления и разместит их в диалоговом окне. Причем это верно для всех окон, создаваемых на базе шаблона блока диалога, — для немодальных блоков диалога и форматных представлений.

Создание элементов управления в тексте программы

Элементы управления можно создать непосредственно в коде программы. Как и другие объекты классов библиотеки MFC, инкапсулирующие визуальные объекты Windows, элементы управления создаются в два этапа: сначала необходимо создать объект данного конкретного класса, а затем вызвать функцию-член *Create* для создания соответствующего элемента управления Windows и связывания его с созданным объектом класса. Это выполняется обычно для блока диалога в функции *OnInitDialog*, а для обычного окна — в функции *OnCreate*.

Следующий пример показывает создание поля редактирования в тексте программы.

Объект *CEdit* объявлен в декларации класса CMarsDialog:

```
class CMarsDialog : public CDialog
{
...

protected:
  CEdit m_edit;

...

public:
  virtual BOOL OnInitDialog();

...
};
```

Затем в функции *OnInitDialog* для него задается прямоугольник, определяющий его размеры и расположение, после чего вызывается функция *Create*

для создания поля редактирования Windows и связывания его с созданным объектом *CEdit*:

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

  ...

  CRect rect(20, 20, 100, 50);
  m_edit.Create(WS_CHILD | WS_VISIBLE | WS_TABSTOP |
  ES_AUTOHSCROLL | WS_BORDER, rect, this, IDC_EDIT);

  m_edit.SetFocus();

  ...

  return FALSE;
}
```

После создания элемента управления функцией *Create* была вызвана функция *SetFocus* для передачи фокуса ввода созданному полю редактирования. И затем функция *OnInitDialog* вернула значение FALSE — в этом случае фокус ввода устанавливается на первый элемент управления.

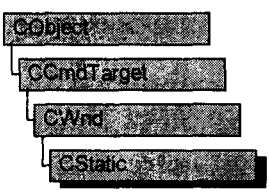
Для создания элементов управления можно в равной мере пользоваться как первым (используя редактор ресурсов) — этот способ применяется чаще, так и вторым способом (создавая элементы управления в коде программы).

Если элемент управления создается в блоке диалога, то он автоматически уничтожается, когда пользователь закрывает диалоговое окно. Если элемент управления создается программно внутри окна, то его необходимо уничтожать. Если он создан с помощью функции *new*, то, соответственно, необходимо вызывать функцию *delete* для уничтожения объекта.

Глава 21

Классы стандартных элементов управления

Статические элементы управления (класс CStatic)



В случаях, когда требуется вывести текст [1] или добавить какие-либо дополнительные рисованные элементы в блоке диалога, функции GDI, как правило, не используются, для этого в Windows существуют статические элементы управления.

Класс *CStatic* позволяет выводить на экран текстовые строки, различные закрашенные прямоугольники, пиктограммы, курсоры, битовые массивы и расширенные метафайлы. Часто они используются для объединения или разделения других элементов управления. Обычно статические элементы управления не требуют ввода и вывода информации, однако они могут уведомлять родительское окно о нажатии клавиши (в случае если они созданы со стилем SS_NOTIFY).

Конструктор

Как и все классы библиотеки MFC, класс *CStatic* имеет конструктор, который, создавая объект класса, никак не воздействует на систему, т. е. не создает объект Windows.

CStatic::**CStatic**()

Инициализация

В отличие от конструктора, функция *Create* создает статический элемент управления Windows и присоединяет его к объекту класса. Функция возвращает значение TRUE в случае удачи и FALSE — в случае ошибки.

---

[1] Например, заголовок другого элемента управления или поясняющий текст.

```
BOOL CStatic::Create(
   LPCTSTR lpszText,
   DWORD dwStyle,
   const RECT& rect,
   CWnd* pParentWnd,
   UINT nID = 0xffff)
```

lpszText — задает текст, который должен быть помещен на элемент управления. Если этот параметр равен NULL, то не будет выведено никакого текста, что вполне естественно, т. к. при задании некоторых стилей статического элемента текст и не нужен.

dwStyle — задает стиль статического элемента. Параметр может содержать комбинацию стилей, объединенных операцией OR. Необходимо всегда указывать стиль WS_CHILD, т. к. элементы управления не могут быть окнами верхнего уровня. Для того чтобы элемент управления был видимым, сразу после отображения его родительского окна необходимо указать стиль WS_VISIBLE. Если нужно, чтобы созданный элемент управления был недоступен для использования, то можно указать стиль WS_DISABLE. Кроме того, для создания требуемого статического элемента необходимо использовать следующие стили:

SS_BLACKFRAME	задает прямоугольник с рамкой того же цвета, что и рамка окна, по умолчанию этот цвет — черный
SS_BLACKRECT	задает прямоугольник, закрашенный цветом, совпадающим с цветом рамки окна, по умолчанию этот цвет — черный
SS_CENTER	определяет прямоугольник и выводит заданный текст в его центре; текст форматируется до того, как он будет выведен на экран; слова, не помещающиеся на одной строке, автоматически переносятся на следующую, что, однако, не гарантирует отображения всего текста
SS_GRAYFRAME	задает прямоугольник с рамкой того же цвета, что и цвет фона клиентской области родительского окна; для блоков диалога этот цвет по умолчанию серый
SS_GRAYRECT	задает прямоугольник, закрашенный цветом, совпадающим с цветом фона клиентской области родительского окна; для блоков диалога этот цвет по умолчанию серый
SS_ICON	задает пиктограмму для ее отображения в родительском окне; передаваемый текст (параметр lpszText) является именем пиктограммы (но не именем файла), определенной в файле ресурса; параметры nWidth и nHeight при этом игнорируются, размеры пиктограммы определяют размеры элемента управления
SS_LEFT	определяет прямоугольник и выводит текст, выровненный по левому краю; текст форматируется до того, как он будет выведен на экран; слова, не помещающиеся на одной строке, автоматически переносятся на следующую, что, однако, не гарантирует отображения всего текста

SS_LEFTNOWORDWRAP определяет прямоугольник и выводит текст, выровненный по левому краю; символы табуляции заменяются пробелами, но слова автоматически не переносятся на новую строку. В качестве символа переноса используется символ "\n"; текст, не помещающийся в строке, обрезается

SS_NOPREFIX если этот стиль не определен, то Windows интерпретирует символ амперсанта (&) в тексте элемента управления как префикс символа-акселератора; в этом случае, символ "&" удаляется и следующий за ним символ в строке подчеркивается; если статический элемент управления содержит текст, где эта особенность не требуется, то может быть добавлен данный стиль; чаще всего стиль используется, когда имена файлов или другие строки, которые будут выведены в статическом элементе управления, могут содержать символ амперсанта

SS_RIGHT определяет прямоугольник и выводит текст, выровненный по правому краю; текст форматируется до того, как он будет выведен на экран; слова, не помещающиеся в строке, автоматически переносятся на следующую, что, однако, не гарантирует отображения всего текста

SS_SIMPLE задает прямоугольник и выводит строку текста, выровненного по левому краю; строка текста не может быть укорочена или изменена; при этом родительское окно или блок диалога не должны обрабатывать сообщение WM_CTLCOLOR

SS_USERITEM статический элемент управления, определяемый пользователем

SS_WHITEFRAME задает прямоугольник с рамкой, которая по умолчанию белого цвета

SS_WHITERECT задает закрашенный прямоугольник по умолчанию белого цвета

Если необходимо вывести на экран битовый массив, пиктограмму, курсор или метафайл, то нужно использовать следующие стили:

SS_BITMAP стиль для битовых массивов

SS_ICON стиль для курсоров и пиктограмм

SS_ENHMETAFILE стиль для расширенных метафайлов

Для курсоров, битовых массивов или пиктограмм можно использовать также стиль:

SS_CENTERIMAGE располагает изображение в центре статического элемента управления

rect — задает положение и размер статического элемента управления. Это может быть структура типа *RECT* или объект класса *CRect*.

pParentWnd — задает родительское окно объекта *CStatic*, обычно это объект класса *CDialog*. Этот параметр должен быть обязательно определен, т. е. не должен быть равен NULL.

nID — задает идентификатор статического элемента управления.

В следующем фрагменте кода создается объект класса *CStatic*:

```
class CNeptuneDialog : public CDialog
{
...
  protected:
    CStatic m_static;
...
public:
  virtual BOOL OnInitDialog();
...
};
```

Затем вызывается функция *Create* для создания элемента управления и связывания его с членом класса *m_static*:

```
BOOL CNeptuneDialog::OnInitDialog()
{
...
  CDialog::OnInitDialog();

  CRect rect(20, 20, 100, 40);
  m_static.Create("Visual C++ и библиотека классов MFC",
    WS_CHILD | WS_VISIBLE | SS_LEFT, rect, this, IDC_STATIC);
...
}
```

В результате в диалоговой панели будет выведен статический текст "Visual C++ и библиотека классов MFC".

Вывод битового массива

Для работы с битовым массивом предназначены две функции *GetBitmap* и *SetBitmap*.

HBITMAP CStatic::**GetBitmap**() —
возвращает дескриптор битового массива, ранее установленного с помощью SetBitmap, в случае отсутствия связанного битового массива с элементом управления или ошибки возвращается NULL.

HBITMAP CStatic::**SetBitmap**(HBITMAP hBitmap) —
задает дескриптор битового массива для его вывода в рамках статического элемента управления (параметр hBitmap).

Вторая функция возвращает дескриптор предыдущего битового массива, связанного со статическим элементом управления, или NULL, если до этого с ним не был связан битовый массив.

Битовый массив будет автоматически выведен в рамках элемента управления. По умолчанию, его левый верхний угол будет совпадать с левым верхним углом статического элемента управления, а размер элемента управления будет изменен в соответствии с размером изображения. Для элемента, отображающего битовый массив, можно использовать различные стили окон и статических элементов управления, включая следующие:

SS_BITMAP — всегда используется, когда выводится битовый массив

SS_CENTERIMAGE — для центрирования изображения в рамках элемента управления; если изображение больше, чем элемент управления, то оно будет обрезано; если изображение меньше, чем элемент управления, то оставшееся место вокруг изображения будет заполнено цветом первого пикселя левого верхнего угла битового массива

Код создания статического элемента управления в середине окна, может выглядеть примерно так:

```
CStatic bmpStatic;
int CVenusFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
  ...
  CBitmap bitmap;
  HBITMAP bm;
  if (!bitmap.LoadBitmap("IDB_BITMAP"))
  {
    bitmap.GetBitmap(&bm);
    CRect rect(0, 0, lpCreateStruct->cx, lpCreateStruct->cy);
    bmpStatic.Create(NULL, WS_CHILD | WS_VISIBLE |
      SS_BITMAP | SS_CENTERIMAGE, rect, this, IDC_BMPSTATIC);
    bmpStatic.SetBitmap(bm);
  }
  ...
}
```

Вывод пиктограммы

Для работы с пиктограммами существуют две функции-члена *GetIcon* и *SetIcon*.

HICON CStatic::**GetIcon**() —
возвращает дескриптор пиктограммы, ранее установленной функцией SetIcon, или NULL.

HICON CStatic::**SetIcon**(HICON hIcon) —
задает дескриптор пиктограммы, которая будет выведена в рамках статического элемента управления (параметр hIcon).

Вторая функция возвращает дескриптор ранее связанной с элементом управления пиктограммы или NULL, если таковой не было.

Пиктограмма будет автоматически выведена в рамках элемента управления. По умолчанию, ее левый верхний угол будет совпадать с левым верхним углом статического элемента управления, а размер элемента управления будет изменен в соответствии с размером пиктограммы. Для элемента, отображающего пиктограмму, можно использовать различные стили окон и статических элементов управления, включая следующие:

SS_ICON всегда используется при выводе курсоров и пиктограмм

SS_CENTERIMAGE применяется для центровки изображения в рамках элемента управления; если изображение больше, чем элемент управления, то оно будет обрезано; если изображение меньше, чем элемент управления, то пустое место вокруг изображения будет заполнено цветом фона статического элемента управления

Вывод курсора

Для работы с курсором предназначены функции *SetCursor* и *SetCursor*.

HCURSOR CStatic::**GetCursor**() —
возвращает дескриптор изображения курсора, ранее установленного функцией SetCursor, или NULL, если ранее курсор не был установлен.

HCURSOR CStatic::**SetCursor**(HCURSOR hCursor) —
задает дескриптор курсора, который должен изображаться в рамках статического элемента управления (параметр hCursor). Функция возвращает дескриптор ранее установленного курсора и NULL — в противном случае.

Действия по размещению изображения курсора, задаваемых параметров и т. д. полностью аналогичны выводу изображения пиктограмм.

Вывод расширенных метафайлов

Для работы с расширенными метафайлами имеются функции *GetEnhMetaFile* и *SetEnhMetaFile*.

HENHMETAFILE CStatic::**GetEnhMetaFile**() —
возвращает дескриптор расширенного метафайла, ранее установленного функцией SetEnhMetaFile, или NULL, если ранее расширенный метафайл не был установлен.

HENHMETAFILE CStatic::**SetEnhMetaFile**(HENHMETAFILE hMetaFile) —
задает дескриптор расширенного метафайла для его отображения в рамках статического элемента управления (параметр hMetaFile).

Содержимое расширенного метафайла будет автоматически выведено в рамках элемента управления. По умолчанию, его левый верхний угол будет совпадать с левым верхним углом статического элемента управления, а размер элемента управления будет изменен в соответствии с размером содержимого метафайла. Для элемента, отображающего метафайл, можно использовать различные стили окон и статических элементов управления, включая следующий:

SS_ENHMETAFILE всегда используется при выводе расширенных метафайлов

Обработка сообщения WM_CTLCOLOR

Следующий пример иллюстрирует, как в блоке диалога в ответ на сообщение WM_CTLCOLOR изменяется кисть для закрашивания фона статических элементов управления.

Для этого необходимо создать обработчик ON_WM_CTRLCOLOR с помощью мастера *ClassWizard*. Обработчик получает параметр, сообщающий тип элемента управления, для которого запрашиваются параметры рисования. В нашем случае задаем только другую кисть, т. к. единственный статический элемент управления блока диалога не имеет текста, и достаточно лишь заменить цвет и стиль закраски фона.

```
/*************************************************************
Файл StaticDlg.cpp
Copyright (c) 1997 Мешков А., Тихомиров Ю., Шведов Д.
*************************************************************/

// StaticDlg.cpp : implementation file
//

...

BEGIN_MESSAGE_MAP(CStaticDlg, CDialog)
  //{{AFX_MSG_MAP(CStaticDlg)
  ON_WM_PAINT()
  ON_WM_QUERYDRAGICON()
  ON_WM_CTLCOLOR()
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////
// CStaticDlg message handlers

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

  // Set the icon for this dialog.
```

```
  // The framework does this automatically
  // when the application's main window is not a dialog
  SetIcon(m_hIcon, TRUE);  // Set big icon
  SetIcon(m_hIcon, FALSE); // Set small icon

  // m_brush — член данных класса CStaticDlg типа CBrush.
  // Создаем кисть со штриховкой в виде "решетки",
  // штрих белого цвета
  m_brush.CreateHatchBrush(HS_CROSS, RGB(255, 255, 255));

  return TRUE;  // return TRUE unless you set the
                // focus to a control
}
...
// обработчик сообщения WM_CTLCOLOR
HBRUSH CStaticDlg::OnCtlColor(
  CDC* pDC, CWnd* pWnd, UINT nCtlColor)
{
  HBRUSH hbr = CDialog::OnCtlColor(pDC, pWnd, nCtlColor);

  // заменяем кисть только для статических
  // элементов управления
  if (nCtlColor == CTLCOLOR_STATIC)
    hbr = m_brush; // происходит вызов функции-оператора
                   // CBrush::operator HBRUSH()

  return hbr;
}
/*************************************************************
Файл Static.rc
Copyright (c) 1997 Мешков А., Тихомиров Ю., Шведов Д.
*************************************************************/

//Microsoft Developer Studio generated resource script.
//

...

//////////////////////////////////////////////////////////
//
// Dialog
//

IDD_STATIC_DIALOG DIALOGEX 0, 0, 185, 92
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION |
WS_SYSMENU
EXSTYLE WS_EX_APPWINDOW
CAPTION "Static"
FONT 8, "MS Sans Serif"
BEGIN
```

```
  DEFPUSHBUTTON    "OK",IDOK,128,7,50,14
  PUSHBUTTON       "Cancel",IDCANCEL,128,23,50,14
  LTEXT            "",IDC_STATIC,0,0,122,92,
                   WS_BORDER,WS_EX_TRANSPARENT
END
...
```

В результате блок диалога будет иметь следующий вид:

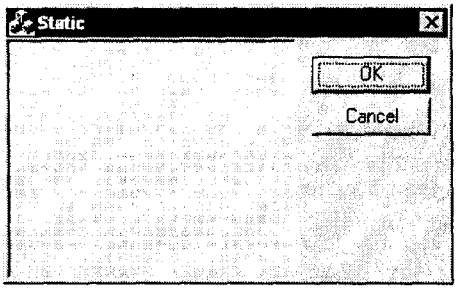
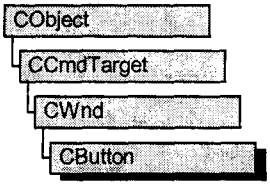


Рис. 16. Блок диалога приложения Static

Кнопки (класс CButton)



Класс *CButton* предназначен для облегчения создания такого элемента управления Windows как "кнопка". Кнопка — это маленькое прямоугольное дочернее окно, кнопка может предоставлять пользователю возможность инициировать определенные приложением действия или позволяет осуществлять выбор. Кнопки могут использоваться по отдельности или группами, могут быть с нанесенным изображением или без него. Кнопка обычно изменяет свой внешний вид, когда пользователь "нажимает" ее.

Типичные примеры кнопок — это командные кнопки, срабатывающие по нажатию (PUSHBUTTON), флажки (CHECK BOX) и переключатели (RADIOBUTTON). Класс *CButton* позволяет создать любую из них, задав

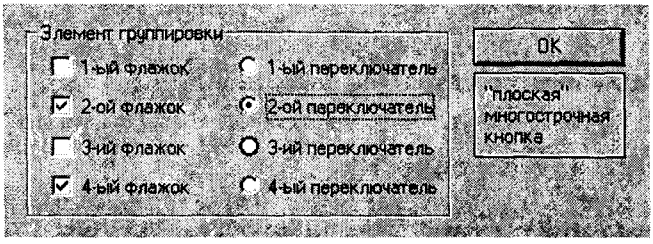


Рис. 17. Кнопки с различными стилями

нужный стиль при создании кнопки с помощью функции *Create*. Кроме типовых кнопок, комбинируя те или иные стили, можно создавать самые разнообразные элементы управления. Таких стилей довольно много, и мы их сгруппировали по назначению.

Вот стили, определяющие типы кнопок:

BS_PUSHBUTTON — определяет простую кнопку, срабатывающую по нажатию; она посылает сообщение WM_COMMAND окну-владельцу, когда пользователь "нажимает".

BS_CHECKBOX — определяет маленький квадратик с расположенным справа текстом (если он не определен вместе со стилем BS_LEFTTEXT, располагающим текст по левой стороне кнопки); такая кнопка может иметь два состояния: отмеченное ("checked" — квадратик помечен крестиком) и неотмеченное ("unchecked" — квадратик пуст)

BS_RADIOBUTTON — определяет небольшую круглую кнопку с расположенным справа от нее текстом (если не включен дополнительно стиль BS_LEFTTEXT, располагающий текст слева); кнопки этого типа обычно используются для реализации выбора одной возможности из нескольких (естественно, для этого следует определить несколько кнопок того же типа)

Кроме создания трех основных типов кнопок есть возможность объединения кнопок в группу, задав следующий стиль:

BS_GROUPBOX — определяет прямоугольную рамку, объединяющую другие кнопки; в левом углу рамки может располагаться текст

Кроме основных вышеперечисленных стилей кнопок существует еще несколько стилей, которые позволяют создать следующие виды кнопок:

BS_DEFPUSHBUTTON — определяет кнопку, выделенную жирной рамкой; при нажатии клавиши <Enter> срабатывает именно эта кнопка; этот стиль позволяет пользователю быстро выбрать наиболее предпочтительный[1] элемент управления в диалоговом окне (команду или кнопку по умолчанию)

BS_AUTOCHECKBOX — то же, что и BS_CHECKBOX, но он автоматически меняет состояние при выборе

BS_AUTORADIOBUTTON — то же, что и BS_RADIOBUTTON, но кнопка-переключатель автоматически помечается при выборе ее пользователем, а с остальных кнопок группы отметка автоматически снимается

[1] С точки зрения разработчика приложения.

BS_3STATE	то же, что и BS_CHECKBOX, но элемент управления имеет три состояния; к двум первым (включен/выключен) добавляется третье, обозначающее некоторое среднее состояние; часто используется в программах-инсталляторах для индикации того, что хотя бы одна составляющая часть программного продукта выбрана для инсталляции
BS_AUTO3STATE	то же, что и BS_3STATE, но смена состояний производится автоматически
BS_LEFTTEXT	может комбинироваться со стилями BS_CHECKBOX и BS_RADIOBUTTON для расположения текста слева от кнопок
BS_OWNERDRAW	задает кнопку, внешний вид которой определяется пользователем; когда необходимо произвести перерисовку кнопки, вызывается функция-член *CButton::DrawItem*. Этот стиль необходимо устанавливать, если используется класс *CBitmapButton*

В Windows 95 и Windows NT определены дополнительные стили для кнопок. Задание их позволяет пользоваться и некоторыми функциями класса *CButton*:

BS_TEXT	кнопка с текстом
BS_ICON	кнопка с пиктограммой или изображением курсора
BS_BITMAP	кнопка с битовым массивом
BS_LEFT	выравнивание по левому краю в области вывода текста или изображения
BS_RIGHT	выравнивание по правому краю в области вывода текста или изображения
BS_CENTER	выравнивание по центру по горизонтали в области вывода текста или изображения
BS_TOP	выравнивание по верхнему краю в области вывода текста или изображения
BS_BOTTOM	выравнивание по нижнему краю в области вывода текста или изображения
BS_VCENTER	выравнивание по центру по вертикали в области вывода текста или изображения
BS_PUSHLIKE	кнопка изображается так же, как обычная кнопка, срабатывающая по нажатию (для флажков или переключателей)
BS_MULTILINE	текст на кнопке может располагаться в несколько строк
BS_NOTIFY	при нажатии или двойном нажатии обычная кнопка посылает уведомляющее сообщение родительскому окну; в рамках библиотеки классов MFC этот стиль не имеет серьезного значения — и то и другое событие можно обрабатывать непосредственно при помощи соответствующих функций (см. ниже)
BS_FLAT	кнопка изображается "плоской" без трехмерной рамки
BS_RIGHTBUTTON	то же, что и BS_LEFTTEXT

Кроме вышеперечисленных стилей следует использовать следующие оконные стили:

WS_CHILD следует указывать всегда при создании кнопки

WS_VISIBLE следует указывать, если кнопка должна быть изначально видимой

WS_DISABLED следует указывать, если кнопка должна быть изначально недоступной

WS_GROUP необходим для объединения кнопок в группу, позволяет переходить от кнопки к кнопке при помощи клавиш перемещения курсора; обязательно должен быть использован для первого переключателя группы таких же переключателей и для следующего после последнего переключателя элемента управления

WS_TABSTOP необходим, если кнопка должна входить в группу кнопок, которым по очереди передается фокус ввода при нажатии клавиши табуляции

Если требуется обрабатывать сообщения, посылаемые кнопкой родительскому окну (обычно это объект класса, производного от *CDialog*), то нужно добавить функцию-обработчик для каждого сообщения в таблицу сообщений.

Для кнопок определены только два вида таких сообщений:

ON_BN_CLICKED посылает родительскому окну сообщение, если пользователь "нажал" кнопку

ON_BN_DOUBLECLICKED посылает родительскому окну сообщение, если пользователь дважды подряд "нажал" кнопку (double click) (речь, естественно, идет о двойном щелчке кнопки мыши на данной кнопке)

Ниже приводится описание функций-членов класса. Следует иметь в виду, что, как обычно, конструктор лишь создает объект класса *CButton*.

Инициализация

```
BOOL CButton::Create(
  LPCTSTR lpszCaption,
  DWORD dwStyle,
  const RECT& rect,
  CWnd* pParentWnd,
  UINT nID) —
```

создает кнопку — элемент управления Windows и присоединяет ее к объекту Cbutton; lpszCaption — задает текст, который будет помещен на кнопке; dwStyle — задает тип кнопки (допускается любая комбинация стилей кнопок[1]); rect — задает размер и расположение кнопки, это может быть объект CRect или

[1] На самом деле задавать сразу все типы кнопок для создаваемого объекта нет смысла. В противном случае либо возникнет ошибка времени выполнения, либо результат — не определен.

структура типа *RECT*; pParentWnd — задает родительское окно кнопки, обычно это объект типа CDialog (этот параметр должен быть обязательно определен, т. е. не должен быть равен NULL); nID — задает идентификатор кнопки.

Функция возвращает TRUE в случае успешного завершения и FALSE — в случае неудачи.

Функции для определения и изменения текущего состояния кнопок

Для определения состояния кнопок предназначены следующие две функции: *GetState* и *SetState*.

Первая, *GetState*, позволяет узнать текущее состояние кнопки любого типа:

UINT CButton::**GetState**() —

возвращает текущее состояние кнопки: выбрана или нет, выделена или нет, имеет фокус ввода или нет.

Кнопки автоматически выделяются, если пользователь нажимает и удерживает левую кнопку мыши в момент, когда курсор мыши находится на кнопке. Выделение с кнопки снимается, как только пользователь отпускает кнопку мыши или перемещает курсор вне кнопки.

Необходимо использовать следующие маски для извлечения информации о состоянии кнопки:

0x0003 для определения того, включена кнопка или нет (только для переключателей и кнопок выбора); значение 0 показывает, что кнопка не включена, 1 — кнопка включена (переключатель включен, если он помечен жирной точкой; флажок включен, если его квадратик перечеркнут крестиком); значение 2 сообщает, что кнопка находится в неопределенном состоянии (только для флажков, имеющих три состояния — флажок в неопределенном состоянии высвечивается серым цветом)

0x0004 для определения того, выделена кнопка или нет (ненулевое значение показывает, что кнопка выделена)

0x0008 для определения состояние фокуса ввода (ненулевое значение означает, что кнопка имеет фокус ввода)

Функция *SetState* позволяет изменять только одно из состояний:

void CButton::**SetState**(BOOL bHighlight) —

устанавливает кнопку в выделенное состояние; bHighlight — определяет, должна ли быть кнопка выделена цветом. Значение TRUE выделяет кнопку, FALSE — снимает выделение кнопки.

Данная функция не влияет на состояние *флажков* и *переключателей*.

Следующие две функции предназначены только для переключателей и кнопок выбора:

int CButton::**GetCheck**() —

возвращает текущее состояние кнопки.

Возвращаемое значение для кнопок, созданных со стилями BS_AUTOCHECKBOX, BS_AUTORADIOBUTTON, BS_AUTO3STATE, BS_CHECKBOX, BS_RADIOBUTTON или BS_3STATE может принимать следующие значения:

0 кнопка не выбрана

1 кнопка выбрана

2 состояние кнопки не определено (только для кнопок имеющих стиль BS_3STATE или BS_AUTO3STATE)

Если кнопка имеет какой-либо другой стиль кроме указанных выше, то возвращается нулевое значение.

void CButton::**SetCheck**(int nCheck) —
устанавливает состояние кнопки (выбрана или нет); nCheck — задает состояние кнопки. Параметр может принимать те же значения, что возвращает функция GetCheck.

Функции для определения и изменения стиля кнопки

Для определения текущих стилей кнопки следует воспользоваться функцией *GetButtonStyle*, которая возвращает только стили, определенные для кнопок, т. е. имеющие префикс BS_.

UINT CButton::**GetButtonStyle**()

А для изменения или установки стилей уже созданной кнопки в классе *CButton* определена функция *SetButtonStyle*:

```
void CButton::SetButtonStyle(
  UINT nStyle,
  BOOL bRedraw = TRUE)
```

где параметр nStyle задает стиль кнопки (см. выше), а параметр bRedraw определяет, должна ли быть кнопка перерисована после изменения стилей (значение TRUE приводит к перерисовке кнопки). По умолчанию параметр принимает значение TRUE, и кнопка будет перерисована немедленно. Возможность отложенной перерисовки позволяет поменять стили сразу нескольких элементов управления, возможно, проводя какие-либо длительные проверки, и только затем одновременно отобразить изменения.

Назначение изображений

Для кнопок, имеющих стили BS_ICON и BS_BITMAP, возможно назначение изображений: пиктограммы, изображения курсоров (BS_ICON) и битовые массивы (BS_BITMAP). По умолчанию пиктограммы, битовые массивы или курсоры помещаются в центре кнопки. Однако, манипулируя соответствующими стилями, эти изображения можно расположить и иначе. Важным моментом является то, что если размер кнопки недостаточен для раз-

мещения пиктограммы, битового массива или курсора, то размещаемое изображение будет обрезано по размерам кнопки.

Все функции этой группы позволяют получать дескрипторы ранее назначенных изображений. Кроме того, отдельно реализованы функции назначения изображений (с префиксом *Set*) и функции только возвращающие дескрипторы (с префиксом *Get*).

Перечислим функции назначения или связывания изображений с объектом-кнопкой:

HICON CButton::**SetIcon**(HICON hIcon)

HCURSOR CButton::**SetCursor**(HCURSOR hCursor)

HBITMAP CButton::**SetBitmap**(HBITMAP hBitmap)

где параметры hIcon, hCursor и hBitmap являются дескрипторами соответствующих типов.

В отличие от класса *CBitmapButton*, который позволяет задавать до четырех изображений на одну кнопку для различных ее состояний, класс *CButton* использует только одно — когда кнопка нажимается, изображение сдвигается вниз и вправо.

Переопределяемые функции

Если при создании кнопки был задан стиль BS_OWNERDRAW, библиотека классов требует обязательного создания производного класса от класса *CButton*, в котором переопределена функция *DrawItem*. Дело в том, что наличие этого стиля при перерисовке объекта-кнопки приводит к вызову функции *CButton::DrawItem* или той же функции в переопределенном классе. Отсутствие переопределения функции приведет к аварийному завершению приложения в момент первой же попытки отображения кнопки. Связано это с тем, что реализация функции *DrawItem* просто генерирует ошибку:

```
void CButton::DrawItem(LPDRAWITEMSTRUCT)
{
  ASSERT(FALSE);
}
```

Все это происходит, однако, только в том случае, если объект *CButton* или производный от него связан с объектом Windows. Иначе кнопка просто не будет отображена — нет обработчика, нет и кнопки.

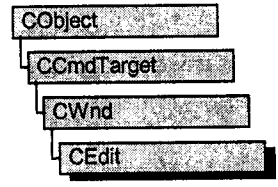
В заключение приведем определение функции, в которой происходит рисование пользовательской кнопки:

virtual void CButton::**DrawItem**(
LPDRAWITEMSTRUCT lpDrawItemStruct)

где параметр lpDrawItemStruct является указателем на структуру *DRAWITEMSTRUCT*. Эта структура содержит информацию о том, что и как

должно быть нарисовано.[1] В одном из полей этой структуры передается дескриптор контекста устройства, для которого следует вызывать функции рисования. Следует иметь в виду, что приложение должно восстановить все использовавшиеся объекты графического интерфейса (GDI), выбранные для этого контекста устройства, перед тем, как функция DrawItem будет завершена.

Простейший текстовый редактор (класс CEdit)[2]



Класс *CEdit* позволяет создать такой элемент управления Windows, как "элемент редактирования", который является простейшим редактором текста. Он представляет собой прямоугольное дочернее окно, в которое может вводиться текст. Когда фокус ввода принадлежит редактору текста, пользователь может вводить в него текст, редактировать его, перемещать курсор по тексту, выделять текст, копировать его в буфер обмена и обратно. Используя класс *CEdit*, можно создать как многострочный редактор текста, так и просто поля для ввода текста.

В большинстве случаев, особенно когда редактор используется для реализации поля ввода, он функционирует автономно — приложение запрашивает введенный текст на момент запроса, никак не контролируя сам процесс ввода информации. Однако редактор предусматривает возможность уведомления (извещения) родительского окна о некоторых событиях. Правильнее следует говорить не о возможности уведомления, а о возможности обработки уведомлений. Для связывания функций обработки предусмотрены макросы, добавляющие обработчик в карту сообщений родительского окна, созданного на базе производных классов от класса *CWnd*. Все эти макросы перечислены ниже совместно с описанием соответствующих уведомляющих сообщений:

ON_EN_CHANGE — пользователь произвел действие, которое, возможно, изменило текст в редакторе; в отличие от сообщения EN_UPDATE, это сообщение посылается после того, как Windows уже обновила окно редактора

ON_EN_ERRSPACE — элемент управления не может получить достаточно памяти для размещения текста

ON_EN_HSCROLL — пользователь произвел щелчок мышью на горизонтальной полосе прокрутки редактора; родительское окно уведомляется до того, как будет произведено обновление экрана

---

[1] Более подробно о структуре *DRAWITEMSTRUCT* см. "Переопределяемые функции" в разделе, посвященном спискам и классу *CListBox*.

[2] Пример использования простейшего редактора можно найти в части X этого тома.

ON_EN_KILLFOCUS редактор теряет фокус ввода

ON_EN_MAXTEXT текущая вставка превышает заданное число символов для редактора и поэтому она усечена; сообщение посылается, если редактор не имеет стиля ES_AUTOHSCROLL, и число символов для вставки превышает ширину редактора; также посылается, если редактор не имеет стиля ES_AUTOVSCROLL, и высота общего числа строк с текстом для вставки превышает высоту окна редактора

ON_EN_SETFOCUS посылается, когда редактор получает фокус ввода

ON_EN_UPDATE сообщает, что редактор текста собирается отобразить измененный текст на экране; посылается после того, как элемент управления отформатировал текст, но еще не вывел изменения на экране (с тем, чтобы можно было в случае необходимости изменить размеры окна)

ON_EN_VSCROLL пользователь щелкнул мышью на вертикальной полосе прокрутки; родительское окно уведомляется до того, как экран будет перерисован

Инициализация

Использование класса *CEdit* может понадобиться как для элементов редактирования, созданных с помощью шаблона блока диалога, так и при чисто программной реализации элемента управления. Однако для создания собственно элементов управления требуется вызвать функцию *Create*:

```
BOOL CEdit::Create(
  DWORD dwStyle,
  const RECT & rect,
  CWnd* pParentWnd,
  UINT nID)
```

которая создает простейший текстовый редактор Windows и присоединяет его к объекту CEdit. В случае успешного завершения функция возвращает значение TRUE и FALSE — в случае неудачи.

Параметр *dwStyle* задает стили создаваемого текстового редактора. Можно комбинировать следующие стили:

ES_AUTOHSCROLL редактор автоматически прокручивает текст вправо на 10 символов, если пользователь вводит символ в конце строки; если пользователь нажимает клавишу ввода, то происходит переход к началу следующей строки

ES_AUTOVSCROLL автоматически прокручивает текст на одну строку, если пользователь нажал клавишу ввода на последней строке

ES_CENTER центрирует текст в многострочном редакторе

ES_LEFT выравнивает текст по левому краю

ES_LOWERCASE все символы, набираемые в редакторе, преобразуются в строчные

ES_MULTILINE задает многострочный редактор (по умолчанию редактор создается как однострочный); если задан стиль ES_AUTOVSCROLL, то редактор прокручивает при необходимости текст по вертикали; если же стиль ES_AUTOVSCROLL не задан, то редактор показывает столько строк, сколько помещается в окне, а затем сообщает звуковым сигналом, если нажимается клавиша ввода, но уже нет места в окне для вывода строк; если задан стиль ES_AUTOHSCROLL, то многострочный редактор автоматически прокручивает текст по горизонтали, когда каретка достигает правого края; для того чтобы начать новую строку, пользователь должен нажать клавишу ввода; если стиль ES_AUTOHSCROLL не задан, то редактор автоматически переносит слова на начало новой строки; перенос слов определяется размерами окна; если размеры окна изменяются, то текст выводится заново с учетом нового размера окна; многострочный редактор может иметь полосы прокрутки; редактор с полосой прокрутки сам обрабатывает сообщения от собственной полосы прокрутки; редактор без полосы прокрутки осуществляет прокрутку, как описано выше, и обрабатывает сообщения для прокрутки, посылаемые ему родительским окном

ES_NOHIDESEL редактор снимает выделение с текста, если он теряет фокус ввода, и вновь его устанавливает при получении; задание стиля ES_NOHIDESEL отключает эту возможность, включенную по умолчанию

ES_OEMCONVERT текст, вводимый в редактор, конвертируется из набора символов ANSI в набор OEM, при передаче его обратно из редактора текст конвертируется в набор ANSI; это обеспечивает правильное преобразование символов, когда приложение вызывает функцию Win32 API ::AnsiToOem для преобразования строки ANSI в строку символов OEM; эта функция может быть полезна для редакторов, содержащих имена файлов

ES_PASSWORD все символы, набираемые в редакторе изображаются на экране звездочками (*); приложение может использовать функцию *CEdit::SetPasswordChar* для изменения символов, которые выводятся на экран

ES_RIGHT выравнивает текст по правому краю в многострочном редакторе

ES_UPPERCASE все символы, набираемые в редакторе, преобразуются в прописные

ES_READONLY предотвращает изменение текста в редакторе (текст предназначен только для чтения, но не для редактирования)

ES_WANTRETURN устанавливает режим, в котором при нажатии клавиши ввода должен вставляться символ возврата каретки; этот стиль не действует в однострочном редакторе; отсутствие этого стиля приведет при нажатии клавиши ввода к выполнению команды, заданной по умолчанию; чтобы этого избежать следует нажимать комбинацию клавиш <Ctrl>+<Enter>

Кроме вышеперечисленных можно использовать следующие стили:

WS_CHILD	следует указывать всегда при создании текстового редактора
WS_VISIBLE	следует указывать, если текстовый редактор должен быть изначально видимым
WS_DISABLED	следует указывать, если текстовый редактор должен быть изначально недоступным
WS_GROUP	необходим для объединения элементов управления в группу[1]
WS_TABSTOP	если текстовый редактор должен входить в группу элементов управления, которым по очереди передается фокус ввода при нажатии клавиши табуляции

Параметр *rect* задает размер и расположение окна редактора. Может быть объектом *CRect* или структурой *RECT*. Параметр *pParentWnd* задает родительское окно (обычно это *CDialog*). Значение этого параметра не должно быть равно NULL. Параметр *nID* задает идентификатор редактора.

Выделение текста

Для того чтобы определить положение текущего выделения текста, в редакторе существуют следующие функции:

```
DWORD CEdit::GetSel() —

void CEdit::GetSel(
   int& nStartChar,
   int& nEndChar)
```

Версия, возвращающая значение DWORD, возвращает начальную позицию выделенного текста в младшем слове и позицию первого невыделенного, следующего за выделенным текстом, символа в старшем слове.

Вторая версия функции имеет параметры *nStartChar* и *nEndChar*, которые являются ссылками на переменные целого типа, принимающие позиции первого выделенного и первого невыделенного символов соответственно.

Для того чтобы изменить выделение текста, в редакторе существуют функции:

```
void CEdit::SetSel(
  DWORD dwSelection,
  BOOL bNoScroll = FALSE)

void CEdit::SetSel(
  int nStartChar,
  int nEndChar,
  BOOL bNoScroll = FALSE)
```

---

[1] Что вряд ли имеет смысл — до окна редактора, например, от кнопки перемещения можно "добраться", но "уйти" от него — нет, т. к. этот элемент управления самостоятельно обрабатывает нажатия клавиш перемещения курсора.

dwSelection — в младшем слове этого параметра задается начальная позиция, в старшем слове — конечная позиция для выделения. Если в младшем слове содержится 0 и в старшем −1, то весь текст в редакторе будет выделен. Если в младшем слове содержится −1, то выделение текста в редакторе, если оно есть, будет снято.

bNoScroll — флаг, определяющий, должна ли быть каретка (маркер ввода с клавиатуры) установлена в зону видимости, возможно при помощи прокрутки текста в окне редактора. Если он равен FALSE, то каретка устанавливается в зону видимости, если TRUE — то нет.

nStartChar — задает начальную позицию. Если *nStartChar* =0 и *nEndChar* =−1, то весь текст в редакторе будет выделен. Если *nStartChar*=−1, то все выделение текста в редакторе будет снято.

nEndChar — задает конечную позицию.

void CEdit::**ReplaceSel**(LPCTSTR lpszNewText) —
функция замещает текущий выделенный текст в редакторе на заданный текст.

lpszNewText — указывает на строку, содержащую текст, на который производится замена.

Если в момент вызова функции текст в редакторе не выделен, то замещающий текст будет вставлен, начиная с текущей позиции каретки. Этой функцией следует пользоваться для замены части текста, если нужно заменить весь текст в редакторе, то лучше использовать функцию *CWnd::SetWindowText*.

Работа со строками

int CEdit::**GetFirstVisibleLine**() —
функция возвращает индекс самой верхней видимой строки в редакторе. Для однострочного редактора возвращается ноль.

void CEdit::**LineScroll**(
int nLines,
int nChars = 0) —
прокручивает текст в многострочном редакторе. nLines — задает число строк, которые должны быть прокручены по вертикали; nChars — задает число позиций символов для прокрутки по горизонтали. Этот параметр игнорируется когда редактор имеет стили ES_RIGHT или ES_CENTER. Знак этих параметров указывает направление: отрицательные значения приводят к прокрутке вверх и влево, а положительные — вниз и вправо соответственно.

Данная функция работает только для многострочного редактора. Она не прокручивает по вертикали текст дальше последней строки в редакторе. Если сумма номера текущей строки и числа строк для прокрутки, заданного параметром *nLines*, превышает общее число строк в редакторе, то значение устанавливается таким образом, что последняя строка текста устанавливается вверху окна редактора.

Функция может быть использована для прокрутки текста по горизонтали за последний символ любой строки.

int CEdit::**LineFromChar**(int nIndex = -1) —

позволяет получить номер строки (параметр nIndex), содержащей заданный *символьный индекс*, где под символьным индексом понимается число символов от начала текста, содержащегося в редакторе.

Если параметр nIndex равен -1, то возвращается номер строки, содержащей первый символ выделенного текста. Если же в редакторе нет выделенного текста, то в этом случае будет возвращен номер текущей строки (строки, на которой находится каретка). Эта функция также предназначена только для многострочного редактора.

int CEdit::**LineLength**(int nLine = -1) —

определяет длину строки, содержащейся в редакторе. Когда функция вызывается для многострочного редактора, возвращается длина строки (в байтах), заданной параметром nLine. Когда функция вызывается для однострочного редактора, возвращается длина текста (в байтах), содержащегося в редакторе. Если параметр nLine равен -1, то возвращается длина текущей строки (содержащей каретку), не включая длину любого выделенного в строке текста. Когда функция вызывается для однострочного редактора, этот параметр игнорируется.

Для получения номера строки в многострочном редакторе следует использовать функцию *CEdit::LineIndex*.

Для получения общего количества строк в редакторе следует воспользоваться функцией *GetLineCount*, которая возвращает единицу в случае, если редактор пуст.

Следующие функции позволяют получить строку из редактора текста.

```
int CEdit::GetLine(
  int nIndex,
  LPTSTR lpszBuffer)
```

и

```
int CEdit::GetLine(
  int nIndex,
  LPTSTR lpszBuffer,
  int nMaxLength)
```

Обе функции возвращают число реально скопированных байт текста и 0, если номер строки, заданный в nIndex, превышает количество строк в редакторе. Параметр nIndex задает номер строки, которую необходимо получить из редактора. Нумерация строк начинается с нуля. Параметр lpszBuffer указывает на буфер, в который будет скопирована строка, причем первое слово буфера должно содержать число, равное максимальному числу байт, которые можно скопировать в буфер. Вторая версия функции помещает в буфер это число самостоятельно на основании значения, заданного параметром nMaxLength. Обратите внимание, что строка, скопированная в буфер, *не будет* содержать символ окончания строки.

Чтобы получить номер первого символа строки от начала буфера, следует воспользоваться функцией *CEdit::LineIndex*. Эта функция работает только в многострочном редакторе.

Для непосредственного доступа к блоку памяти, в котором редактор хранит отображаемый текст, класс *CEdit* имеет две функции — первая для получения дескриптора блока, а вторая — для назначения блока редактору. Рассмотрим их подробнее.

HLOCAL CEdit::**GetHandle**() —

эта функция возвращает дескриптор памяти, выделенной для многострочного редактора. В случае ошибки, например, при ее вызове для однострочного редактора, возвращается ноль. Следует вызывать данную функцию для многострочного редактора, имеющего в качестве родительского окна блок диалога со стилем DS_LOCALEDIT. Если же блок не имеет такого стиля, значение, возвращаемое функцией, не будет являться дескриптором, и соответственно им нельзя будет пользоваться.

void CEdit::**SetHandle**(HLOCAL hBuffer) —

устанавливает дескриптор локальной памяти (параметр hBuffer), которая будет использоваться многострочным редактором текста. Этот дескриптор должен быть создан путем вызова функции Win32 API ::LocalAlloc с использованием флага LMEM_MOVEABLE. Блок памяти должен содержать строку, заканчивающуюся нулевым символом.

Буфер, полученный при вызове этой функции, редактор будет использовать для хранения текста взамен своего собственного буфера. До того как приложение установит новый блок памяти, необходимо использовать функцию GetHandle для получения дескриптора текущего буфера памяти и освободить эту память с помощью функции Win32 API ::LocalFree. Функция SetHandle очищает буфер отмены последней операции (теперь функция CanUndo возвращает 0) и сбрасывает флаг модификации (функция GetModify возвращает 0). Окно редактора перерисовывается.

Эту функцию можно использовать в диалоговом блоке многострочного редактора только в том случае, если диалоговое окно было создано со стилем DS_LOCALEDIT.

Работа с блоками теста

void CEdit::**Clear**() —

удаляет выделенный текст (если он есть).

Для восстановления удаленного текста (откат назад) можно вызвать функцию-член *CEdit::Undo*. Для удаления текущего выделения в редакторе и одновременного помещения его буфер обмена следует воспользоваться функцией *CEdit::Cut*.

void CEdit::**Copy**() —

копирует выделенный текст (если он есть) в буфер обмена в формате CF_TEXT.

void CEdit::**Cut**() —

удаляет (вырезает) выделенный текст в редакторе (если он есть) и копирует удаленный текст в буфер обмена в формате CF_TEXT.

void CEdit::**Paste**() —

вставляет данные из буфера обмена в редактор с текущей позиции курсора. Данные вставляются только в том случае, если буфер обмена содержит данные в формате CF_TEXT.

BOOL CEdit::**Undo**() —

отменяет последнюю операцию, произведенную в редакторе (откат на один шаг назад). Для однострочного редактора возвращаемое значение всегда TRUE. Для многострочного редактора возвращается TRUE в случае успешного завершения и FALSE, если операция отката завершилась неуспешно.

Над операцией отката также можно произвести операцию отката. Например, если некоторый текст в редакторе был удален, а затем восстановлен вызовом функции *CEdit::Undo*, то, вызвав еще раз *CEdit::Undo*, можно опять удалить текст.

BOOL CEdit::**CanUndo**() —

позволяет определить, может ли быть отменена последняя операция, совершенная в редакторе, или нет. Значение TRUE, возвращенное функцией, означает, что последняя операция, совершенная в редакторе, может быть отменена вызовом функции CEdit::Undo.

void CEdit::**EmptyUndoBuffer**() —

сбрасывает флаг отмены последней операции в редакторе. В этом случае отмена последней выполненной операции становится невозможной. Флаг отмены последней операции устанавливается всякий раз, когда действие операции внутри редактора может быть отменено. Флаг отмены последней операции автоматически сбрасывается всякий раз, когда вызываются функции CWnd::SetWindowText и CEdit::SetHandle.

Параметры и флаги

Для того чтобы определить, изменялся ли текст в редакторе, имеется функция *GetModify*:

BOOL CEdit::**GetModify**()

которая возвращает значение TRUE, если содержимое редактора изменилось, и FALSE — в противном случае.

Windows поддерживает внутренний флаг (флаг модификации), показывающий, менялось ли содержимое редактора. Этот флаг сбрасывается когда редактор создается, он также может быть сброшен (или установлен) функцией *SetModify*:

void CEdit::**SetModify**(BOOL bModified = TRUE) —

устанавливает (bModified = TRUE) или сбрасывает (bModified = FALSE) флаг модификации.

Следующая функция позволяет установить или отменить режим редактора "только для чтения", при котором можно только просматривать текст в редакторе.[1]

---

[1] При этом цвет фона клиентской области окна становится таким же, как и цвет трехмерных объектов, например, кнопок. Однако, в отличие от состояния, при котором элемент является недоступным (стиль WM_DISABLE), цвет символов текста тот же, что и в доступных для изменения редакторах.

BOOL CEdit::**SetReadOnly**(BOOL bReadOnly = TRUE)

Функция возвращает значение TRUE в случае успешного установления требуемого режима и FALSE — в случае неудачи. Параметр bReadOnly устанавливает (TRUE) или снимает (FALSE) режим "только для чтения".

Текущее состояние можно узнать, проанализировав состояние флага ES_READONLY, вызвав функцию *CWnd::GetStyle* и проанализировав возвращенное ею значение.

void CEdit::**GetRect**(LPRECT lpRect) —

позволяет получить границы форматирования редактора, которые представляют собой прямоугольник, размеры которого не зависят от размера окна редактора.

Размеры прямоугольника, в пределах которого форматируется текст в многострочном редакторе, могут быть изменены вызовом функций *CEdit::SetRect* и *CEdit::SetRectNP*.

void CEdit::**LimitText**(int nChars = 0) —

ограничивает длину текста, который пользователь может ввести в окне редактора. Параметр nChars задает длину (в байтах) текста, который может ввести пользователь. Если параметр равен нулю, то длина текста устанавливается равной UINT_MAX (максимальное значение, которое может быть сохранено в переменной типа unsigned int). Это же значение задано по умолчанию.

Функция задает длину текста, который пользователь может ввести, но она не влияет на длину теста, уже находящегося в редакторе. Также она не влияет на длину текста, копируемого функцией *CWnd::SetWindowText*.

Если прикладная программа использует функцию *SetWindowText* для того, чтобы поместить в редактор большее количество текста, чем задано функцией *LimitText*, пользователь сможет только удалять часть текста (модификация и удаление исключаются) в редакторе.

В Windows 95 и Windows NT рекомендуется использовать функцию *CEdit::SetLimitText*.

При создании приложений, в которых пользователю требуется ввести с клавиатуры какой-либо пароль, могут быть полезны функции, позволяющие вместо вводимых с клавиатуры символов выводить на экран последовательность той же длины, состоящую из одного и того же символа (по умолчанию это символ "*"). Класс *CEdit* имеет в своем составе функцию для получения установленного в данный момент символа замены (*GetPasswordChar*) и функцию назначения такого символа (*SetPasswordChar*). Эти функции не действуют для многострочных редакторов.

Функции только для Windows 95

Для определения и изменения полей редактора предназначены следующие две функции:

DWORD CEdit::**GetMargins**() —

возвращает значения границ полей редактора в пикселях, ширина левого края содержится в младшем слове, ширина правого — в старшем.

void CEdit::**SetMargins**(
UINT nLeft,
UINT nRight) —
устанавливает значения границ полей, параметры nLeft и nRight определяют новую ширину полей в пикселях.

Для определения и изменения максимального размера текста, который может содержаться в редакторе, предназначены функции *GetLimitText* и *SetLimitText*.

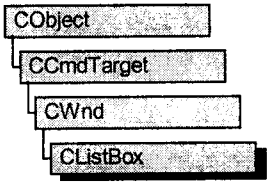
Для определения номера символа и номера строки по указанным координатам предназначена функция *CharFromPos*:

int CEdit::**CharFromPos**(CPoint pt) —
номер символа возвращается в младшем слове, а номер строки — в старшем.

Следующая функция позволяет произвести обратную операцию: по номеру символа определить его координаты:

CPoint CEdit::**PosFromChar**(UINT nChar) —
Если значение параметра nChar больше, чем количество символов, содержащихся в редакторе, то будут возвращены координаты последнего символа.

Список (класс CListBox)



С помощью класса *CListBox* можно создать такой элемент управления Windows, как "список". Список представляет собой прямоугольник, внутри которого находится последовательность тестовых или пользовательских элементов, которые пользователь может пролистывать и выбирать. Список может использоваться, например, для просмотра имен файлов на диске.

Можно создавать списки, позволяющие выбирать только одну запись (элемент) или несколько одновременно (списки множественного выбора). Когда пользователь выбирает запись в списке — она выделяется, и окно-список посылает родительскому окну извещение. Вообще, список посылает или, правильнее сказать, может посылать несколько извещений родительскому окну. Для них всех предусмотрены перечисленные ниже специальные макросы, позволяющие добавить в карту сообщений окна функции обработки. Функции, добавленные первыми тремя макросами, будут вызваны только для списков, содержащих текстовые строки и созданных со стилем (о которых чуть позже) LBS_NOTIFY. Большинство уведомлений посылается родительскому окну тогда и только тогда, когда список является текущим элементом управления.

ON_LBN_DBLCLK	добавляет обработчик двойного щелчка кнопки мыши в области списка
ON_LBN_SELCANCEL	добавляет обработчик снятия выбора с элементов списка

ON_LBN_SELCHANGE добавляет обработчик смены выбора элемента списка, однако он не вызывается при смене выбора функцией *CListBox::SetCurSel*; для списков множественного выбора обработчик вызывается при любом нажатии клавиш управления курсором, независимо от того была ли смена выбора

ON_LBN_SETFOCUS добавляет обработчик получения списком фокуса ввода

ON_WM_CHARTOITEM добавляет обработчик уведомляющего сообщения для самоотображаемых списков (owner-draw), не имеющих строковых элементов при получении ими сообщения WM_CHAR

ON_WM_VKEYTOITEM добавляет обработчик уведомляющего сообщения для списков, имеющих стиль LBS_WANTKEYBOARDINPUT, при получении ими сообщения WM_KEYDOWN

ON_LBN_ERRSPACE добавляет обработчик уведомления о невозможности получить достаточно запрашиваемой памяти

ON_LBN_KILLFOCUS добавляет обработчик потери фокуса ввода списком

Окно-список можно создавать как в шаблоне блока диалога, так и непосредственно в коде программы.

Инициализация

Создание объекта *CListBox* и собственно списка Windows в коде программы требует создания (вызова) конструктора класса и функции *Create*:

```
BOOL CListBox::Create(
   DWORD dwStyle,
   const RECT & rect,
   CWnd* pParentWnd,
   UINT nID)
```

которая создает окно-список и присоединяет его к объекту CListBox. Она возвращает значение TRUE, если завершилась успешно и FALSE в противном случае.

Параметр *dwStyle* задает стиль списка, который может принимать комбинацию следующих значений:

LBS_EXTENDEDSEL пользователь может выбирать несколько записей одновременно, используя клавишу <Shift> и мышь или специальную комбинацию клавиш

LBS_HASSTRINGS задает список, который может содержать произвольные строки; приложение может заполнять строками список и извлекать из него строки при помощи функции *CListBox::GetText*

LBS_MULTICOLUMN определяет список, отображаемый в несколько столбцов, который может осуществлять горизонтальную прокрутку; ширина всех столбцов в пикселях устанавливается функцией *CListBox::SetColumnWidth*

LBS_MULTIPLESEL	создает список множественного выбора, в котором при выборе элемента его состояние меняется (выделен — не выделен), не влияя на состояние других элементов
LBS_NOINTEGRALHEIGHT	размер окна списка точно соответствует размеру, указанному приложением при его создании; иначе Windows подгоняет его размер, с тем, чтобы показывалось целое число элементов
LBS_NOREDRAW	изменения, производимые со списком, не отражаются автоматически в окне; этот стиль может быть изменен в любой момент посылкой сообщения WM_SETREDRAW
LBS_NOTIFY	родительское окно будет получать извещения при возникновении ряда событий (см. выше)
LBS_OWNERDRAWFIXED	отображение содержимого окна возлагается на функцию родительского окна, при этом элементы списка имеют одинаковую высоту
LBS_OWNERDRAWVARIABLE	отображение содержимого окна возлагается на функцию родительского окна, при этом элементы списка могут иметь разную высоту
LBS_SORT	строки списка сортируются по алфавиту
LBS_STANDARD	строки списка сортируются по алфавиту и родительское окно получает извещение, если пользователь производит одинарный или двойной щелчок на строке списка; сам список окружен рамкой
LBS_USETABSTOPS	при отображении строковых элементов используются символы табуляции; по умолчанию позиция табуляции задается равной 32 диалоговым единицам; диалоговые единицы вычисляются как четверть базовой ширины, возвращаемой функцией Win32 API *::GetDialogBaseUnits*
LBS_WANTKEYBOARDINPUT	родительское окно списка получает сообщение WM_VKEYTOITEM или WM_CHARTOITEM, если пользователь нажимает клавишу в момент, когда список имеет фокус ввода
LBS_DISABLENOSCROLL	список показывает скрытую вертикальную полосу прокрутки, даже когда список не имеет достаточного количества записей для прокрутки в окне

Кроме вышеперечисленных стилей следует использовать следующие стили:

WS_CHILD	следует указывать всегда при создании списка
WS_VISIBLE	следует указывать, если список изначально должен быть видимым
WS_DISABLED	следует указывать, если список изначально должен быть недоступным

WS_GROUP	необходим для объединения списков в группу с другими элементами
WS_TABSTOP	необходим, если кнопка должна входить в группу кнопок, которым по очереди передается фокус ввода при нажатии клавиши табуляции
WS_VSCROLL	необходим для добавления вертикальной полосы прокрутки
WS_HSCROLL	необходим для добавления горизонтальной полосы прокрутки

Параметр *rect* задает размер и расположение списка на экране, параметр *pParentWnd* задает родительское окно (обычно объект *CDialog*), который *не может* быть равен NULL, а параметр *nID* задает идентификатор списка.

Изменение параметров списка

Не все параметры списка можно задать при помощи редактора ресурсов или функции *Create*. Далее приведены описания функций-членов класса *CListBox*, использование которых позволяет настраивать список в соответствии с требованиями приложения.

Для изменения ширины списка предназначены следующие две функции (обратите внимание, что речь идет не о размерах окна, ограничивающего список, а о поле списка, только часть которого может быть видима "через" это окно):

int CListBox::**GetHorizontalExtent**() —

возвращает ширину списка в пикселях, в пределах которой список может прокручиваться по горизонтали. Функцию можно использовать только для списков, имеющих горизонтальную полосу прокрутки.

void CListBox::**SetHorizontalExtent**(int cxExtent) —

устанавливает ширину списка по горизонтали (в пикселях), в пределах которой список может прокручиваться. Если размер окна списка меньше этого значения, то строки в списке можно будет прокручивать с помощью горизонтальной полосы прокрутки. Если окно списка такого же размера или больше, то горизонтальная полоса прокрутки делается невидимой. Для того чтобы список реагировал на SetHorizontalExtent, нужно, чтобы он был создан со стилем WS_HSCROLL. Для списков, имеющих несколько столбцов, вместо этой функции следует использовать функцию CListBox::SetColumnWidth.

Для получения размера прямоугольника, который ограничивает текущую изображаемую запись списка, предназначена функция

```
int CListBox::GetItemRect(
  int nIndex,
  LPRECT lpRect)
```

где nIndex задает индекс записи, а lpRect является указателем на структуру *RECT*, возвращающую координаты прямоугольника, в который выводится запись списка. В случае ошибки возвращает LB_ERR.

Следующая пара функций позволяет получить и изменить высоту записей в списке:

int CListBox::**GetItemHeight**(int nIndex) —

возвращает высоту записи в списке в пикселях. Если список создан с использованием стиля LBS_OWNERDRAWVARIABLE, то функция возвращает высоту записи списка с указанным индексом (параметр nIndex). В противном случае возвращается LB_ERR.

int CListBox::**SetItemHeight**(
int nIndex,
UINT cyItemHeight) —

устанавливает высоту записи в списке, где nIndex задает индекс записи в списке. Этот параметр используется лишь в том случае, если список имеет стиль LBS_OWNERDRAWVARIABLE, иначе он должен быть равен нулю. Параметр cyItemHeight задает высоту записи в пикселях. Функция возвращает LB_ERR, если индекс или высота записи имеют неверное значение.

Для изменения ширины столбцов предназначена функция *SetColumnWidth*:

void CListBox::**SetColumnWidth**(int cxWidth) —

устанавливает ширину столбцов (в пикселях) в списке, имеющем несколько столбцов (созданном со стилем LBS_MULTICOLUMN), где cxWidth задает ширину в пикселях всех столбцов.

Имеется возможность задания позиций табуляции в списке. Для этого предназначены следующие функции:

void CListBox::**SetTabStops**() —

вариант функции, не имеющий аргументов, устанавливает позиции табуляции по умолчанию (в 32 диалоговые единицы).

BOOL CListBox::**SetTabStops**(const int& cxEachStop) —

функция для установки значения позиции табуляции, заданной параметром cxEachStop. Данное значение позиции табуляции устанавливается для всех одно и отсчитывается от предыдущей позиции.

Напротив, следующая функция позволяет указывать различные значения позиций для нескольких табуляций:

BOOL CListBox::**SetTabStops**(
int nTabStops,
LPINT rgTabStops) —

позиции табуляции будут установлены в соответствии с каждым элементом массива rgTabStops, размер которого задан параметром nTabStops. Значения позиций табуляций отсчитываются от левого края окна списка, таким образом все элементы массива как бы отсортированы по возрастанию. Для возможности использования функций SetTabStops список должен быть создан с использованием стиля LBS_USETABSTOPS.

Функции возвращают значение TRUE, если все позиции табуляции установлены, иначе — FALSE.

Для того чтобы правильно обрабатывать строки, необходимо задание параметра, идентифицирующего локализацию. Для этого предназначен идент-

фикатор локализации LCID, представляющий собой 32-битовое значение, идентифицирующее язык локализации программы, которое определяет таблицы символов для хранения и отображения текстовых ресурсов, правила сортировки строк, раскладку клавиатуры, форматы даты и времени. Это необходимо, например, для правильной сортировки строк в списке. Для определения и изменения этого идентификатора служат следующие две функции:

LCID CListBox::**GetLocale**() —
возвращает идентификатор локализации для данного списка.

LCID CListBox::**SetLocale**(LCID nNewLocale) —
устанавливает идентификатор локализации для списка, где nNewLocale — новый локальный идентификатор для списка.

Если функция *SetLocale* не вызывалась, то значение LCID по умолчанию берется из операционной системы. Значение по умолчанию в системе может изменяться путем изменения установок с помощью программ настройки Windows (установка языка и национальных стандартов).

Функции для работы с содержимым списка

После того как список был создан, необходимо его заполнить. Следующие функции предназначены для работы со строками списка. Они позволяют выполнить такие основные операции, как добавление и удаление строк из списка, вставку строк в определенное место в списке, поиск строк, удаление всех строк, а также заполнение списка имен файлов. К этой же группе относятся функции получения информации о конкретных элементах списка и ассоциирования с ними дополнительных данных. Последняя возможность позволяет создавать списки объектов самых разных типов, возлагая при этом работу по отображению их описания на Windows, исключая таким образом необходимость создания функций отображения элементов, что нужно для самоотображаемых элементов управления.

Итак, начнем с наиболее часто используемой функции, которая добавляет строку текста в качестве элемента в список:[1]

int CListBox::**AddString**(LPCTSTR lpszItem) —
функция возвращает индекс новой строки в списке. Параметр lpszItem указывает на завершающуюся нулевым символом строку, которая должна быть добавлена к списку.

Если список не имеет стиля LBS_SORT, то строка будет добавлена в конец списка. Если же этот стиль был задан при создании списка, то строка будет добавлена в список, и список будет заново отсортирован. Если же список

[1] Все функции добавления элементов в список в случае ошибки возвращают значение LB_ERR, а если недостаточно места для хранения новой строки, то возвращают LB_ERRSPACE, при этом может генерироваться соответствующее уведомление.

имеет стиль LBS_SORT, но не имеет стиля LBS_HASSTRINGS, то список будет сортироваться путем многократного вызова функции *CListBox::CompareItem*, которую требуется переопределить, создав соответствующий производный класс.

Списки, не имеющие стиля LBS_SORT, могут использовать функцию *InsertString* для вставки строкового элемента в заданную позицию:

int CListBox::**InsertString**(
  int nIndex,
  LPCTSTR lpszItem) —

функция возвращает индекс новой строки в списке. Кстати сказать, если список имеет стиль LBS_SORT, то значение параметра nIndex, который и определяет желаемую позицию строки, может не соответствовать возвращаемому значению функции; что вполне естественно. Если nIndex = -1, то строка добавляется в конец списка. Параметр lpszItem является указателем на оканчивающуюся нулевым символом строку, которая должна быть вставлена в список.

Очень часто в приложениях списки применяются для того, чтобы вывести список файлов на диске, для их просмотра и/или выбора.[1] Для того чтобы облегчить заполнение списка в этих случаях, существует функция *Dir*:

int CListBox::**Dir**(
  UINT attr,
  LPCTSTR lpszWildCard) —

функция заполняет список содержимым директориев. Параметр attr определяет, файлы с какими атрибутами будут добавлены в список. Он может быть любой комбинацией из следующих значений:

0x0000	файлы, доступные для чтения и записи
0x0001	файлы, доступные только для чтения
0x0002	скрытые файлы
0x0004	системные файлы
0x0010	подкаталоги
0x0020	файлы с установленным битом архивации
0x4000	включать имена дисков
0x8000	специальный флаг; если он установлен, то только файлы указанного типа (заданного в *lpszWildCard*) включаются в список; если не установлен, то файлы заданного типа включаются в дополнение к остальным файлам

Параметр lpszWildCard является указателем на строку, специфицирующую файлы. Строка может содержать символы "*" или "?", которые имеют тот же смысл, что принят в MS-DOS (например, *.cpp, *.c?). К сожалению, поддержка шаблонов, принятых в UNIX, не реализована.

---

[1] Приводим описание функции для лучшего понимания возможностей библиотеки классов MFC, да и Win32 API, т. к. и сама библиотека (см. главу 15 первого тома книги, класс *CFileDialog*) и собственно Windows (см. главу 6 первого тома книги, приложение-пример APIWin) имеют средства отображения файлов и каталогов (папок, если хотите).

Если необходимо добавить в список большое количество строк (например, больше 100), то лучше воспользоваться следующей функцией:

int CListBox::**InitStorage**(
int nItems,
UINT nBytes)

которая позволяет выделить память для хранения строк списка, определяемую с помощью значений параметров nItems (количество записей) и nBytes (размер памяти в байтах на один элемент). Эта функция позволяет ускорить инициализацию списка с большим количеством записей. При ее применении выполнение функций CListBox::AddString, CListBox::InsertString и CListBox::Dir занимает минимальное время. Если при вызове функции памяти будет запрошено меньше, чем нужно для размещения всех строк, то будет использовано обычное выделение памяти для записей, которым не хватило выделенного объема.

Как и любой список вообще, "список" — элемент управления Windows и вместе с ним класс *CListBox*, требуют, наряду с функциями добавления/заполнения элементов, и функции удаления/очистки.

int CListBox::**DeleteString**(UINT nIndex) —

эта функция удаляет элемент, определяемый индексом (параметр nIndex), из списка и возвращает количество оставшихся элементов. Значение LB_ERR возвращается, если заданный индекс записи на удаление больше числа записей в списке.

void CListBox::**ResetContent**() —

эта функция удаляет все элементы из списка.

Следующие две функции предназначены для поиска строки в списке, совпадающей, начиная с первого символа, с заданной строкой.

int CListBox::**FindString**(
int nStartAfter,
LPCTSTR lpszItem) —

осуществляет только поиск строки в списке.

int CListBox::**SelectString**(
int nStartAfter,
LPCTSTR lpszItem) —

ищет и, кроме того, выделяет найденную строку, причем, если строка найдена, то она выделяется и, если нужно, список прокручивается с тем, чтобы строка стала видна.

Обе функции возвращают индекс найденной строки в списке или LB_ERR в случае неудачи. Параметр *nStartAfter* должен содержать индекс строки списка, находящейся перед строкой, с которой будет начат поиск. Если достигнут конец списка, то поиск будет продолжен с начала списка до строки, заданной в *nStartAfter*. Если *nStartAfter* = −1, то поиск будет осуществляться по всему списку от его начала. Параметр *lpszItem* является указателем на оканчивающуюся нулевым символом строку, которая должна быть найдена, причем поиск идет без учета регистра.

Вторая из описанных функций не может использоваться для списков, созданных со стилем LBS_MULTIPLESEL.

Для поиска строк, кроме того, существует еще одна функция, но в отличие от предыдущих она осуществляет поиск строки в списке, *точно* совпадающей с заданной строкой:

```
int CListBox::FindStringExact(
   int nIndexStart,
   LPCTSTR lpszFind)
```

Назначение параметра *nIndexStart*, несмотря на название, полностью аналогично назначению параметра *nStartAfter* предыдущих функций. Если список создан как самоотображаемый, но без стиля LBS_HASSTRINGS, то функция *FindStringExact* попытается согласовать 32-битовое значение, связанное с элементом, со значением параметра *lpszFind* (указатели в Win32 API 32-битные).

Операции над элементами списка

Функции этой группы предназначены для получения информации о конкретных элементах списка или, наоборот, для выделения элементов списка и связывания с ними 32-битных значений и указателей.

Естественно, чтобы работать со списком любого вида, требуется знать, сколько элементов он содержит. Для этого в классе *CListBox* предназначена функция *GetCount*:

```
int CListBox::GetCount()
```

которая возвращает либо количество элементов в списке либо LB_ERR в случае ошибки.

Для получения информации о том, выбран ли определенный элемент или нет, следует использовать функцию *GetSel*, которая может работать как со списками, позволяющими выбирать одну запись (списки одиночного выбора), так и со списками, позволяющими выбирать сразу несколько записей (списки множественного выбора).

Для получения индекса текущей выбранной записи и, наоборот, задания конкретной записи в качестве выбранной следует использовать функции *GetCurSel*, *SetCurSel*, *GetSelItems*, *SetSel*.[1] Первые две функции предназначены, прежде всего, для списков одиночного выбора, а вторые две — для списков множественного выбора. Однако нет никаких ограничений по использованию функций со стороны библиотеки классов и Windows, и не удивляйтесь, если функции будут вести себя неадекватно ситуации. Например, функция *GetCurSel* возвращает значение LB_ERR как в случае отсутствия выбранного элемента, так и в случае ее вызова для списка множественного выбора, независимо от того, есть фактически выбранные элементы или нет. Наиболее интересная из перечисленных — функция *GetSelItems*:

---

[1] Выбор элемента приведет при необходимости к прокручиванию списка таким образом, чтобы выбранный элемент стал видимым в окне списка.

int CListBox::**GetSelItems**(
int nMaxItems,
LPINT rgIndex) —

позволяет получить индексы строк, которые выбраны в данный момент в списке, и это значение может быть больше, чем размерность буфера (параметр rgIndex), заданная параметром nMaxItems. Причем, при использовании этой функции для списка одиночного выбора она возвратит значение LB_ERR.

В добавление к описанным функциям, класс *CListBox* содержит функцию (*SetItemRange*), позволяющую выбирать несколько записей, следующих подряд.

Для получения содержимого текстовых элементов списка класс *CListBox* предусматривает целый набор функций. Во-первых, для надежных программ, которые не полагаются на предположение о длине запрашиваемых данных, существует функция *GetTextLen*, которая возвращает длину строкового элемента в байтах, исключая завершающий нуль-символ. Во-вторых, для получения собственно текста следует использовать функцию *GetText*, предоставляющую буфер для копирования в него символов.[1]

С первого взгляда может показаться, что элемент управления "список" предназначен, в основном, для текстовых элементов или для визуализации в виде списка произвольных данных, скорее, временного использования. В отличие от чисто "программных" классов-контейнеров описываемые списки отличаются в сторону визуализации и обладают меньшей гибкостью, требуемой для хранения объектов самых разных типов. Да, действительно, элемент управления "список" существует только тогда, когда он виден или, по крайней мере, может немедленно быть отображенным.

Тем не менее, и описываемые списки имеют средства для ассоциирования элементов списка с любыми объектами, как пользовательскими, так и системными. Для этого списки поддерживают возможность хранения 32-разрядных чисел или указателей для каждого элемента списка. Причем даже в случае изменения индекса элемента, что может происходить вследствие добавления, удаления и пересортировки списка, значение, связанное с конкретным элементом, будет связано именно с ним, независимо от индекса элемента в списке.

Перечислим функции, реализующие возможность как связывания чисел (указателей) с конкретным элементом, так и обратной операции: *GetItemData, SetItemData, GetItemDataPtr, SetItemDataPtr.*

В ряде случаев бывает необходимо определить, какой из элементов является видимым. Для определения и изменения первой видимой записи списка предназначены функции *GetTopIndex* и *SetTopIndex.*

---

[1] Эта функция имеет версию, где в качестве приемника указывается ссылка на объект класса *CString*. Использование этой функции исключает необходимость самостоятельного определения размера строкового элемента, выделения памяти под принимающий буфер, а затем и освобождения памяти.

И последняя функция этой группы, позволяющая определить, около какого элемента списка находится курсор мыши:

UINT CListBox::**ItemFromPoint**(
CPoint pt,
BOOL & bOutside) —

возвращает индекс записи списка, ближайшей к указанной точке, заданной параметром pt, причем координаты указаны относительно левого верхнего угла клиентской области окна списка. Параметр bOutside — ссылка на переменную типа BOOL, которая по завершении функции будет установлена в TRUE, если указанная точка лежит вне клиентской области окна списка, и в FALSE, если точка находится внутри клиентской области окна списка.

Переопределяемые функции

Если создается список, внешний вид которого определяется пользователем (списки со стилями LBS_OWNERDRAWFIXED или LBS_OWNERDRAWVARIABLE), то *необходимо* переопределить следующие две функции.

Функция, вызываемая для перерисовки элементов списка:

virtual void CListBox::**DrawItem**(
LPDRAWITEMSTRUCT lpDrawItemStruct) —

вызывается библиотекой, если необходима перерисовка элемента самоотображаемого списка, внешний вид которого определяется пользователем. Параметр lpDrawItemStruct является указателем на структуру *DRAWITEMSTRUCT*, которая содержит информацию о типе требуемой перерисовки.

Рассматривая класс *CButton*, мы уже упоминали эту структуру. Пришло время рассмотреть ее подробнее.

Структура *DRAWITEMSTRUCT* имеет следующее определение:

```
typedef struct tagDRAWITEMSTRUCT {
  UINT CtlType;
  UINT CtlID;
  UINT itemID;
  UINT itemAction;
  UINT itemState;
  HWND hwndItem;
  HDC hDC;
  RECT rcItem;
  DWORD itemData;
} DRAWITEMSTRUCT;
```

Рассмотрим ее члены.

CtlType — тип самоотображаемого элемента управления, внешний вид которого определяется пользователем:

ODT_BUTTON	кнопка
ODT_COMBOBOX	комбинированный список
ODT_LISTBOX	список
ODT_MENU	меню

CtlID — идентификатор кнопки, списка или комбинированного списка. Для меню этот параметр не используется.

itemID — идентификатор пункта меню или индекс записи в списке или комбинированном списке. Для списка или комбинированного списка, не содержащих записей, этот параметр должен иметь отрицательное значение, в результате чего будет выведен прямоугольник с координатами, заданными в *rcItem*, и не содержащий никаких элементов.

itemAction — определяет, как должна осуществляться перерисовка. Он может быть комбинацией следующих битов:

ODA_DRAWENTIRE	весь элемент управления требует перерисовки
ODA_FOCUS	устанавливается, если элемент управления получает или теряет фокус ввода; поле *itemState* должно быть проверено для определения, имеет ли элемент управления фокус ввода
ODA_SELECT	устанавливается, если сделан выбор; поле *itemState* должно быть использовано для определения нового выбора

itemState — задает состояние записи после перерисовки:

ODS_DISABLED	запись недоступна
ODS_FOCUS	запись имеет фокус ввода
ODS_SELECTED	запись выбрана

hwndItem — дескриптор окна элемента управления. *hDC* — контекст устройства вывода. *rcItem* — прямоугольник, задающий границы перерисовки. *itemData* — для списка или комбинированного списка этот параметр содержит значение, установленное функциями *CComboBox::AddString*, *CComboBox::InsertString*, *CListBox::AddString* или *CListBox::InsertString*, а для меню — функциями *CMenu::AppendMenu*, *CMenu::InsertMenu* или *CMenu::ModifyMenu*.

Функция, вызываемая для определения размеров элементов списка:

virtual void CListBox::**MeasureItem**(
LPMEASUREITEMSTRUCT lpMeasureItemStruct) —

вызывается при создании списка, внешний вид которого определяется пользователем, для определения размеров элемента списка. Параметр lpMeasureItemStruct является указателем на структуру *MEASUREITEMSTRUCT*.

По умолчанию эта функция не производит никаких действий. Необходимо переопределить эту функцию, в которой заполнить структуру типа *MEASUREITEMSTRUCT* для того, чтобы Windows могла определить размеры списка. Если список создан со стилем LBS_OWNERDRAWVARIABLE, то

эта функция вызывается для каждой записи списка. Если нет, то функция будет вызвана всего один раз.

Структура *MEASUREITEMSTRUCT* имеет следующий вид:

```
typedef struct tagMEASUREITEMSTRUCT {
  UINT CtlType;
  UINT CtlID;
  UINT itemID;
  UINT itemWidth;
  UINT itemHeight;
  DWORD itemData
} MEASUREITEMSTRUCT;
```

Рассмотрим ее члены.

CtlType — содержит информацию о типе элемента управления (см. описание структуры *DRAWITEMSTRUCT*). *CtlID* — идентификатор элемента управления. *itemID* — идентификатор элемента в списке (для списка и комбинированного списка). Не используется для списков и комбинированных списков с фиксированной высотой элементов. *itemWidth* — используется для задания ширины элемента в меню (для списков и комбинированных списков не используется). *itemHeight* — задает высоту записи в списке (максимальное значение равно 255). *itemData* — назначение этого поля аналогично одноименному полю структуры *DRAWITEMSTRUCT*.

Функция, вызываемая при сортировке списков, не имеющих стиля LBS_HASSTRINGS и имеющих стиль LBS_SORT, обязательно должна быть переопределена:

virtual int CListBox::**CompareItem**(
  LPCOMPAREITEMSTRUCT lpCompareItemStruct) —
    вызывается библиотекой для определения расположения нового элемента в списке, внешний вид которого определяется пользователем и который должен осуществлять сортировку записей (список имеет стиль LBS_SORT).

Параметр *lpCompareItemStruct* является указателем на структуру *COMPAREITEMSTRUCT*, в которой определяется относительное положение двух сравниваемых элементов списка. Функция возвращает следующие значения:

-1	первый элемент меньше второго
0	первый элемент равен второму
1	первый элемент больше второго

Структура *COMPAREITEMSTRUCT* имеет следующий вид:

```
typedef struct tagCOMPAREITEMSTRUCT {
  UINT CtlType;
  UINT CtlID;
```

```
  HWND hwndItem;
  UINT itemID1;
  DWORD itemData1;
  UINT itemID2;
  DWORD itemData2;
} COMPAREITEMSTRUCT;
```

Рассмотрим ее члены.

CtlType — содержит информацию о типе элемента управления (см. описание структуры *DRAWITEMSTRUCT*). *CtlID* — идентификатор кнопки, списка или комбинированного списка (для меню этот параметр не используется). *hwndItem* — дескриптор окна элемента управления. *itemID1* — индекс в списке первого из сравниваемых элементов. *itemData1* — данные первого элемента. *itemID2* — индекс в списке второго из сравниваемых элементов. *itemData2* — данные второго элемента.

Пример самоотображаемого списка

Рассмотрим пример приложения диалогового типа, которое содержит один элемент управления — "список". Причем этот список создан со стилями LBS_OWNERDRAWVARIABLE и LBS_NOINTEGRALHEIGHT. Для реализации списка, имеющего различные размеры по вертикали для каждого элемента, создается объект класса *CListBox*, у которого реализованы две функции: *DrawItem* и *MeasureItem*. После части исходных текстов приведен пример окна приложения.

Обратите внимание на то, что функция *MeasureItem* "ничего не делает" из-за того, что настройка высоты элемента осуществляется непосредственно перед выводом элемента списка на экран. Заранее получить высоту элементов мы не имеем возможности, т. к. окно приложения и сам список могут изменять размеры по желанию пользователя.

```
/*************************************************************
Файл LBDlg.cpp
Copyright (c) 1997 Мешков А., Тихомиров Ю., Шведов Д.
*************************************************************/

...

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

  ...

  CRect rect;
  GetClientRect(&rect);
  m_List.MoveWindow(&rect);
```

```
  // настраиваем начальные размеры для всех элементов списка
  //  (в нашем примере это делать не обязательно)
  m_List.SetItemHeight(0, 36);

  // добавляем три элемента в список
  m_List.AddString(_T("Короткий текст"));
  m_List.AddString(_T("Средний по длине текст\в две строки"));
  m_List.AddString(_T("Длинная строка, для отображения которой
                       необходимо несколько строк"));

return TRUE;  // return TRUE  unless you set
              the focus to a control
}

...

void CLBDlg::OnSize(UINT nType, int cx, int cy)
{
  CDialog::OnSize(nType, cx, cy);

  // настраиваем размеры списка по
  // размерам окна тогда, когда
  // объект-список уже есть в системе
  if (::IsWindow(m_List.m_hWnd))
  {
    CRect rect(0, 0, cx, cy);

    m_List.MoveWindow(&rect);
    m_List.InvalidateRect(&rect);
  }
}

/*************************************************************
Файл VListBox.cpp
Copyright (c) 1997 Мешков А., Тихомиров Ю., Шведов Д.
*************************************************************/

// VListBox.cpp : implementation file
//

#include "stdafx.h"
#include "LBSmpl.h"
#include "VListBox.h"

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

// минимальная толщина обрамления пиктограммы в элементе списка
#define FRAME  2
```

```
//////////////////////////////////////////////////////
// CVListBox

CVListBox::CVListBox()
{
}

CVListBox::~CVListBox()
{
}

BEGIN_MESSAGE_MAP(CVListBox, CListBox)
  //{{AFX_MSG_MAP(CVListBox)
    // NOTE - the ClassWizard will add and
    //        remove mapping macros here.
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()

//////////////////////////////////////////////////////
// CVListBox message handlers

void CVListBox::DrawItem(LPDRAWITEMSTRUCT lpDrawItemStruct)
{
  CDC dc;
  dc.Attach(lpDrawItemStruct->hDC);

  CRect rect = lpDrawItemStruct->rcItem;

  // вычисляем смещение для вывода текста (32 пикселя -
  // ширина пиктограммы)
  rect.left += 32 + FRAME + FRAME;

  // вычисляем требуемую высоту области вывода текста
  int nHeight = dc.DrawText((LPCTSTR)lpDrawItemStruct->itemData,
    rect, DT_CALCRECT | DT_LEFT | DT_WORDBREAK) + FRAME + FRAME;

  rect.bottom = rect.top + nHeight;

  if (nHeight < 32 + FRAME + FRAME)
    rect.bottom = rect.top + 32 + FRAME + FRAME;

  // изменяем высоту элемента управления
  SetItemHeight(lpDrawItemStruct->itemID, rect.Height());

  // поле структуры необходимо изменить самостоятельно
  lpDrawItemStruct->rcItem.bottom = rect.bottom;
  rect = lpDrawItemStruct->rcItem;

  // определяем состояние элемента списка
  if (lpDrawItemStruct->itemState & ODS_SELECTED)
  {
    // рисуем "утопленную" кнопку
    dc.DrawFrameControl(&rect, DFC_BUTTON,
```

```
      DFCS_BUTTONPUSH | DFCS_PUSHED);
  }
  else
  {
    // рисуем "выпуклую" кнопку
    dc.DrawFrameControl(&rect, DFC_BUTTON, DFCS_BUTTONPUSH);
  }

  // рисуем пиктограмму, центрируя ее по вертикали
  rect.top += (rect.Height() - 32) / 2;
  dc.DrawIcon(rect.left+FRAME, rect.top,
    AfxGetApp()->LoadIcon(IDR_MAINFRAME));

  rect = lpDrawItemStruct->rcItem;
  rect.left += 32 + FRAME;

  rect.DeflateRect(FRAME, FRAME);
  rect.top += (rect.Height() - (nHeight - FRAME)) / 2;

  // выводим текст элемента управления
  dc.SetBkMode(TRANSPARENT);
  dc.DrawText((LPCTSTR)lpDrawItemStruct->itemData, rect,
    DT_LEFT | DT_WORDBREAK);

  dc.Detach();
}

// эту функцию необходимо было перегрузить, но поскольку
// она вызывается только при добавлении элемента в список,
// а в нашем примере высота элемента настраивается при каждом
// отображении, следовательно, функция ничего не делает
void CVListBox::MeasureItem(
  LPMEASUREITEMSTRUCT lpMeasureItemStruct)
{
}
```

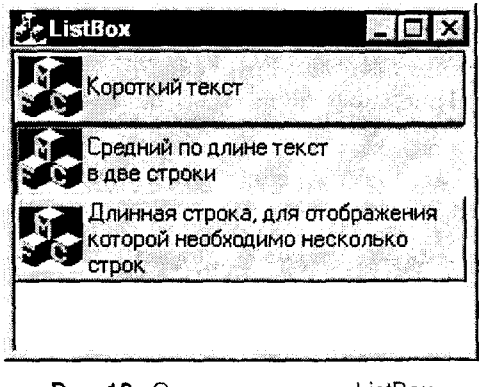
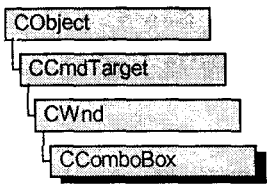


Рис. 18. Окно приложения ListBox

Комбинированный список (класс CComboBox)



Ранее были рассмотрены классы *CEdit* и *CListBox*, позволяющие создавать элементы управления "редактор текста" и "список", но они настолько часто применяются в паре друг с другом, что в Windows был создан элемент управления, сочетающий свойства обоих элементов. Этот элемент представляет собой комбинацию окна списка с окном текстового редактора и носит название "комбинированный список". Часть комбинированного списка — непосредственно список — может выводиться на экран все время (стиль CBS_SIMPLE) или может как бы выпадать из окна редактирования (стили CBS_DROPDOWN и CBS_DROPDOWNLIST). Текущая выбранная запись списка выводится в прямоугольном окне, которое представляет собой окно простейшего текстового редактора и может быть доступна (стили CBS_SIMPLE и CBS_DROPDOWN) или недоступна (стиль CBS_DROPDOWNLIST) для редактирования. Если к тому же комбинированный список определен как раскрывающийся, то при нажатии символьной клавиши на клавиатуре строка, начинающаяся с данного символа, будет выделена.

Если вы хотите обрабатывать уведомляющие сообщения Windows, посылаемые комбинированным списком своему родительскому окну (обычно это объект класса, производного от *CDialog*), то необходимо добавить в карту сообщений этого окна соответствующую функцию-обработчик для этих сообщений. Далее перечислены макросы, с помощью которых и производится назначение функций для обработки сообщений:

ON_CBN_CLOSEUP — окно списка в комбинированном списке закрыто; это сообщение не посылается для комбинированного списка имеющего стиль CBS_SIMPLE

ON_CBN_DBLCLK — пользователь сделал двойной щелчок мышью на строке списка в комбинированном списке; это сообщение посылается только для комбинированного списка, имеющего стиль CBS_SIMPLE; для комбинированного списка со стилями CBS_DROPDOWN или CBS_DROPDOWNLIST двойной щелчок не имеет смысла, т. к. уже одинарный щелчок мышью приводит к сворачиванию списка

ON_CBN_DROPDOWN — список комбинированного списка развернут; это сообщение может быть послано комбинированным списком, имеющим стиль CBS_DROPDOWN или CBS_DROPDOWNLIST

ON_CBN_EDITCHANGE — пользователь совершил действия, которые привели к изменению текста в редакторе комбинированного списка; в отличие от сообщения CBN_EDITUPDATE это сообщение посылается после того, как Windows обновила экран; это сообщение не посылается, если комбинированный список имеет стиль CBS_DROPDOWNLIST

ON_CBN_EDITUPDATE	редактор комбинированного блока собирается вывести на экран измененный текст; сообщение посылается после того, как текст был отформатирован, но перед тем, как он будет выведен на экран; сообщение не посылается, если комбинированный список имеет стиль CBS_DROPDOWNLIST
ON_CBN_ERRSPACE	комбинированный список не может получить достаточно памяти для выполнения запроса
ON_CBN_SELENDCANCEL	показывает, что выбор, сделанный пользователем, должен быть отменен; пользователь щелкнул мышью на записи блока и затем на другом окне или элементе управления; сообщение посылается до сообщения CBN_CLOSEUP для уведомления, что выбор пользователя должен быть игнорирован
ON_CBN_SELENDOK	пользователь выбрал строку списка и затем нажал клавишу <Enter> или щелкнул на кнопке комбинированного списка со стрелкой вниз для того, чтобы список был бы свернут; сообщение посылается до сообщения CBN_CLOSEUP для того, чтобы выбор строк пользователем был учтен; сообщения CBN_SELENDCANCEL и CBN_SELENDOK посылаются даже в случае, если сообщение CBN_CLOSEUP не посылается
ON_CBN_SELCHANGE	выбор в списке комбинированного списка будет изменен в результате щелчка мыши на списке или нажатия клавиш перемещения курсора; при обработке этого сообщения текст из комбинированного списка может быть получен только с помощью функции *CComboBox::GetLBTex;* функция *GetWindowText* использоваться не может
ON_CBN_SETFOCUS	комбинированный список получил фокус ввода
ON_CBN_ERRSPACE	элемент управления не может получить достаточно памяти для размещения текста или элементов списка

Инициализация

Создание объекта *CComboBox* и собственно комбинированного списка Windows в коде программы требует создания (вызова) конструктора класса и функции *Create*:

```
BOOL CComboBox::Create(
  DWORD dwStyle,
  const RECT & rect,
  CWnd* pParentWnd,
  UINT nID) —
```

создает комбинированный список и связывает его с объектом CComboBox. Функция возвращает значение TRUE в случае успешного завершения. Параметр dwStyle задает стиль комбинированного списка, который может принимать комбинацию следующих значений:

CBS_AUTOHSCROLL	текст в редакторе автоматически прокручивается по горизонтали вправо и влево, если пользователь набирает символы в конце строки; если этот стиль не задан, то невозможно ввести текст, длина которого превышает размеры окна редактора
CBS_SIMPLE	список изображается на экране все время, текущий выделенный элемент списка отображается в окне редактирования; редактирование разрешено
CBS_DROPDOWN	подобен стилю CBS_SIMPLE за исключением того, что список не изображается до тех пор, пока пользователь не нажмет кнопку со стрелкой вниз, расположенную справа от окна редактирования комбинированного списка. При этом список "выпадает" из строки редактирования (раскрывающийся список); редактирование разрешено
CBS_DROPDOWNLIST	подобен CBS_DROPDOWN, но вместо строки редактирования имеется строка со статическим текстом, который отображает текущий выбор, но редактировать его нельзя
CBS_HASSTRINGS	комбинированный список, определяемый пользователем; на него накладывается ограничение: строки, помещаемые в список, должны быть текстовыми; комбинированный список выделяет память и хранит указатели на строки; приложение может использовать функцию-член *CComboBox::GetText* для получения текста текущего выбора
CBS_OEMCONVERT	набор символов текста, вводимого в редактор, конвертируется из ANSI в OEM и затем, при передаче данных из комбинированного списка, — обратно в ANSI; это обеспечивает правильное преобразование символов, когда приложение вызывает функцию Win32 API *::AnsiToOem* для преобразования строки символов ANSI в строку символов OEM; эта функция может быть полезна для комбинированных списков, содержащих имена файлов, и применима только в случае, если используются стили CBS_SIMPLE или CBS_DROPDOWN
CBS_OWNERDRAWFIXED	разработчик приложения сам отвечает за перерисовку содержимого элемента управления, при этом все элементы списка имеют одинаковую высоту
CBS_OWNERDRAWVARIABLE	разработчик приложения сам отвечает за перерисовку содержимого элемента управления, при этом все элементы списка могут иметь различную высоту

CBS_SORT	строки, вводимые в список, сортируются автоматически
CBS_DISABLENOSCROLL	при установке этого стиля список выводит на экран полосу вертикальной прокрутки даже если нет достаточного количества элементов для прокрутки в окне (в этом случае полоса прокрутки видна, но недоступна для использования)
CBS_NOINTEGRALHEIGHT	задание этого стиля требует, чтобы размер комбинированного списка точно соответствовал заданному в приложении при его создании; обычно Windows изменяет размер комбинированного списка так, чтобы в его окне были видны элементы списка целиком

Параметр rect указывает расположение и размер комбинированного списка. Может быть структурой *RECT* или объектом класса CRect. Параметр pParentWnd задает родительское окно (обычно объект CDialog), и *не может* быть равен NULL, а параметр nID задает идентификатор комбинированного списка.

Основные операции с комбинированным списком

Так как комбинированный список представляет собой комбинацию списка и окна редактирования, то он имеет как функции, аналогичные функциям его составных частей (списка и редактора), так и собственные функции.

Многие функции целиком аналогичны по параметрам и действию соответствующим функциям классов *CEdit* и *CListBox*, во избежание повторения приведем здесь только их список и краткое описание.

Функции, аналогичные функциям класса *CEdit*:

DWORD CComboBox::**GetEditSel**() —
возвращает позиции первого и последнего выделенных символов текущей выбранной строки, находящейся в окне редактирования.

void CComboBox::**Clear**() —
удаляет выделенный текст (если он есть).

void CComboBox::**Copy**() —
копирует выделенный текст (если он есть) в буфер обмена в формате CF_TEXT.

void CComboBox::**Cut**() —
удаляет (вырезает) выделенный текст в редакторе (если он есть) и помещает удаленный текст в буфер обмена в формате CF_TEXT.

void CComboBox::**Paste**() —
вставляет данные из буфера обмена в редактор с текущей позиции курсора.

BOOL CComboBox::**LimitText**(int nMaxChars) —
ограничивает количество вводимых символов.

Функции, аналогичные функциям класса *CListBox*:

int CComboBox::**GetCount**() —
возвращает количество элементов в списке комбинированного списка или CB_ERR в случае ошибки.

int CComboBox::**GetCurSel**() —
возвращает индекс текущего выбранного в списке элемента. Если не выбран ни один элемент, то возвращается значение CB_ERR.

int CComboBox::**SetCurSel**(int nSelect) —
выбирает элемент в списке и прокручивает список так, чтобы этот элемент был виден. В случае ошибки возвращает CB_ERR.

DWORD CComboBox::**GetItemData**(int nIndex) —
возвращает 32-битовое значение, ассоциированное с элементом комбинированного списка, или CB_ERR в случае ошибки.

int CComboBox::**SetItemData**(
int nIndex,
DWORD dwItemData) —
ассоциирует 32-битовое значение с указанным элементом комбинированного списка. В случае ошибки возвращает CB_ERR.

void* CComboBox::**GetItemDataPtr**(int nIndex) —
возвращает указатель, ассоциированный с элементом комбинированного списка, или CB_ERR в случае ошибки.

int CComboBox::**SetItemDataPtr**(
int nIndex,
void* pData) —
ассоциирует указатель с заданным элементом комбинированного списка. В случае ошибки возвращает CB_ERR.

int CComboBox::**GetTopIndex**() —
возвращает индекс первой видимой строки списка в комбинированном списке.

int CComboBox::**SetTopIndex**(int nIndex) —
устанавливает в верхней строке списка строку с заданным индексом.

UINT CComboBox::**GetHorizontalExtent**() —
возвращает величину полосы прокрутки (в пикселях), в пределах которой комбинированный список может прокручиваться по горизонтали.

void CComboBox::**SetHorizontalExtent**(UINT nExtent) —
задает величину полосы прокрутки (в пикселях), в пределах которой список может быть прокручен по горизонтали в комбинированном списке.

int CComboBox::**SetItemHeight**(
int nIndex,
UINT cyItemHeight) —
устанавливает высоту в пикселях строки списка (для списков с фиксированными по высоте строками nIndex = 0) или высоту окна редактирования (nIndex = -1).

int CComboBox::**GetItemHeight**(int nIndex) —
возвращает высоту в пикселях строки списка (для списков с фиксированными по высоте элементами nIndex = 0) или высоту окна редактирования (nIndex = -1).

LCID CComboBox::**GetLocale**() —
возвращает идентификатор локализации для данного комбинированного списка.

LCID CComboBox::**SetLocale**(LCID nNewLocale) —
устанавливает идентификатор локализации для комбинированного списка.

int CComboBox::**InitStorage**(
int nItems,
UINT nBytes) —
выделяет память для добавления большого количества строк к комбинированному списку.

Кроме общих с классами *CEdit* и *CListBox* функций комбинированный список имеет и свои специфические функции.

Для управления раскрывающимся списком предназначены следующая группа функций:

void CComboBox::**ShowDropDown**(BOOL bShowIt = TRUE) —
делает видимым или невидимым список в комбинированном списке, имеющем стили CBS_DROPDOWN или CBS_DROPDOWNLIST.

BOOL CComboBox::**GetDroppedState**() —
определяет, видим или нет раскрывающийся список в комбинированном списке.

Следующие две функции позволяют управлять шириной раскрывающегося списка при помощи получения/задания ее минимального допустимого значения. По умолчанию минимальная допустимая ширина списка равна 0. Ширина списка определяется максимальным значением из минимальной допустимой и заданной шириной комбинированного списка. Обе функции применимы лишь для комбинированных списков со стилями CBS_DROPDOWN или CBS_DROPDOWNLIST.

int CComboBox::**GetDroppedWidth**() —
возвращает минимальную допустимую ширину раскрывающей части списка в комбинированном списке.

int CComboBox::**SetDroppedWidth**(UINT nWidth) —
устанавливает минимальную допустимую ширину раскрывающегося списка в комбинированном списке.

void CComboBox::**GetDroppedControlRect**(LPRECT lprect) —
возвращает экранные координаты видимого раскрывающегося списка. lprect — указатель на структуру *RECT*, которая получит запрашиваемые координаты

Для выделения символов в окне редактирования комбинированного списка предназначена функция

BOOL CComboBox::**SetEditSel**(
int nStartChar,
int nEndChar)
nStartChar — задает начальную позицию (если задан равным -1, то любое существующее выделение будет снято); nEndChar — задает конечную позицию (если

задан равным -1, то весь текст от начальной позиции до конца будет выделен; конечная позиция задает положение первого невыделенного символа после выделения); в случае неудачного завершения функция возвращает CB_ERR, значение CB_ERR будет также возвращено, если комбинированный список имеет стиль CBS_DROPDOWNLIST или не имеет элементов.

Следующие две функции

```
int CComboBox::GetLBText(
  int nIndex,
  LPCTSTR lpszText)
```

и

```
void CComboBox::GetLBText(
  int nIndex,
  CString & rString) —
```

возвращают строку из списка комбинированного списка. Если параметр nIndex задает несуществующий номер элемента, то возвращается CB_ERR; lpszText — указатель на буфер, получающий строку (буфер должен иметь достаточно места для размещения строки и символа конца строки); rString — ссылка на объект CString (вторая версия функции заполняет этот объект текстом полученной строки).

Для определения длины строки (в байтах) из комбинированного списка используется функция:

```
int CComboBox::GetLBTextLen(int nIndex)
```

Для раскрывающегося списка могут использоваться два типа интерфейса: стандартный и расширенный.

BOOL CComboBox::**GetExtendedUI**() —

определяет, какой интерфейс используется (стандартный или расширенный). Значение TRUE означает расширенный интерфейс.

Расширенный интерфейс имеет следующие отличия:

- щелчок мышью на статическом тексте списка приводит к раскрытию списка (только для комбинированных списков, имеющих стиль CBS_DROPDOWNLIST),
- нажатие клавиши на клавиатуре приводит к раскрытию списка.

int CComboBox::**SetExtendedUI**(BOOL bExtended = TRUE) —

задает, должен ли использоваться интерфейс по умолчанию или расширенный интерфейс пользователя для комбинированных списков со стилями CBS_DROPDOWN или CBS_DROPDOWNLIST, параметр bExtended задает интерфейс: TRUE определяет расширенный интерфейс, FALSE — стандартный.

Операции со строками и виртуальные функции

Функции работы со строками в списке комбинированного списка и виртуальные функции целиком аналогичны соответствующим функциям класса *CListBox* (см. описание класса), поэтому ниже приводится только их список.

int CComboBox::**AddString**(LPCTSTR lpszString) —
добавляет строку в список.

int CComboBox::**DeleteString**(UINT nIndex) —
удаляет строку из списка.

int CComboBox::**InsertString**(
int nIndex,
LPCTSTR lpszString) —
вставляет строку в список.

void CComboBox::**ResetContent**() —
удаляет все элементы списка.

int CComboBox::**Dir**(
UINT attr,
LPCTSTR lpszWildCard) —
заполняет список именами файлов.

int CComboBox::**FindString**(
int nStartAfter,
LPCTSTR lpszString) —
осуществляет поиск строки.

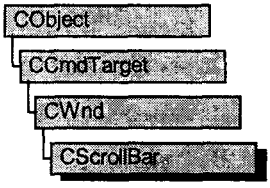
int CComboBox::**FindStringExact**(
int nIndexStart,
LPCTSTR lpszFind) —
ищет строку, целиком совпадающую с заданной.

int CComboBox::**SelectString**(
int nStartAfter,
LPCTSTR lpszString) —
выбирает строку.

virtual void CComboBox::**DrawItem**(LPDRAWITEMSTRUCT lpDrawItemStruct) —
вызывается при выводе на экран комбинированного списка, внешний вид которого определяется пользователем.

virtual void CComboBox::**MeasureItem**(LPMEASUREITEMSTRUCT lpMeasureItemStruct) —
вызывается для определения размеров строки в комбинированном списке, внешний вид которого определяется пользователем.

Полоса прокрутки (класс CScrollBar)



Рассмотрим еще один элемент управления — полосу прокрутки (SCROLL BAR). В Windows существует два варианта использования полос прокрутки. Первый — когда полоса прокрутки функционирует как часть обычного окна или блока диалога. Второй — когда она может действовать, как любой другой элемент управления Windows. И хотя в обоих случаях управление полосой прокрутки происходит одинаково, все же использование ее как самостоятельного элемента управления требует применения некоторых специальных приемов.

Полосы прокрутки бывают вертикальные и горизонтальные, каждая полоса прокрутки имеет кнопки со стрелками, нажимая на которые можно управлять полосой прокрутки, а также бегунок, передвигая его с помощью мыши также можно управлять полосой прокрутки.

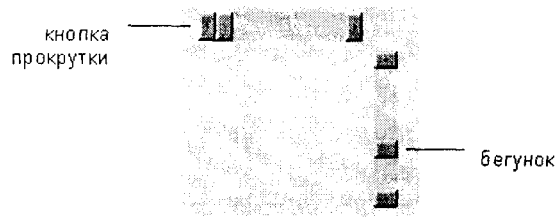


Рис. 19. Полосы прокрутки

В отличие от других элементов управления, полосы прокрутки посылают уведомления о различных событиях и сменах состояния в рамках одного сообщения. Точнее, таких сообщений два — по одному для горизонтальной и вертикальной полос прокрутки. Чтобы назначить свои собственные обработчики сообщений полос прокрутки, следует при помощи средства ClassWizard создать функции *OnHScroll* и *OnVScroll* для класса, определяющего родительское окно. Мы приводим описание этих функций для класса *CWnd*, а не для некоторого пользовательского класса окна. Дело в том, что, действительно, класс *CWnd*, как и некоторые другие оконные классы библиотеки MFC, содержит соответствующие обработчики. Однако эти функции — не виртуальные. Сделано так для того, чтобы исключить влияние со стороны разработчика приложения в случае, если возникнет необходимость вызова этих функций базового класса для обработки именно тех данных, которые передает элемент управления при уведомлении.

Реализовать это можно, повторно запросив параметры сообщения при помощи функции *CWnd::GetCurrentMessage*.

Для обработки сообщений от горизонтальной полосы прокрутки вызывается функция *OnHScroll*:

```
void CWnd::OnHScroll(
  UINT nSBCode,
  UINT nPos,
  CScrollBar* pScrollBar)
```

Параметр *nSBCode* содержит код события и может принимать одно из следующих значений:

SB_LEFT	прокрутка до самой левой границы
SB_ENDSCROLL	завершение прокрутки, т. е. взаимодействия пользователя с бегунком
SB_LINELEFT	прокрутка на одну позицию влево

SB_LINERIGHT	прокрутка на одну позицию вправо
SB_PAGELEFT	поблочная прокрутка влево
SB_PAGERIGHT	поблочная прокрутка вправо
SB_RIGHT	прокрутка до самой правой границы
SB_THUMBPOSITION	прокрутка до позиции, определяемой параметром *nPos*
SB_THUMBTRACK	прокрутка осуществляется бегунком, текущее положение которого определяется параметром *nPos*

Параметр *nPos* определяет текущее положение бегунка для кодов прокрутки SB_THUMBPOSITION или SB_THUMBTRACK, иначе не используется. Так как положение бегунка определяется заданными пределами, параметр *nPos* может содержать отрицательные значения, следовательно, при необходимости следует осуществлять преобразование типов от UINT (беззнаковое целое) к int (знаковое целое).

Параметр *pScrollBar* содержит указатель на (возможно) временный объект, в случае, если сообщение пришло не от оконной полосы прокрутки, наличие которой определяется стилями окна. В этом случае параметр равен NULL.

Тип прокрутки определяется установками полосы прокрутки и методом, каким осуществляется прокрутка. Поблочная прокрутка осуществляется при нажатии кнопки мыши в момент, когда курсор мыши находится в области полосы вне кнопок прокрутки или бегунка. Прокрутка на одну позицию осуществляется при нажатии одной из кнопок прокрутки.[1]

В случае обработки сообщений от полосы прокрутки без вызова обработчика по умолчанию (*CWnd::Default*) или той же функции базового класса, можно самостоятельно установить новое положение при помощи функции *CWnd::SetScrollPos* или *CScrollBar::SetScrollBar*.

Для обработки сообщений от вертикальной полосы прокрутки вызывается функция *OnVScroll*:

```
void CWnd::OnVScroll(
  UINT nSBCode,
  UINT nPos,
  CScrollBar* pScrollBar)
```

Параметр *nSBCode* содержит код события и может принимать одно из следующих значений:

SB_BOTTOM	прокрутка до самой нижней границы
SB_ENDSCROLL	завершение прокрутки, т. е. взаимодействия пользователя с бегунком

[1] Дословный перевод этих терминов следующий; позиция называется *строкой*, а блок — *страницей*, что действительно обычно соответствует реализации. Однако для горизонтальной прокрутки сложно говорить о строках, да и о страницах тоже.

SB_LINEDOWN	прокрутка на одну позицию вниз
SB_LINEUP	прокрутка на одну позицию вверх
SB_PAGEDOWN	поблочная прокрутка вниз
SB_PAGEUP	поблочная прокрутка вверх
SB_TOP	прокрутка до самой верхней границы
SB_THUMBTRACK	прокрутка до позиции, определяемой параметром *nPos*
SB_THUMBPOSITION	прокрутка осуществляется бегунком, текущее положение которого определяется параметром *nPos*

Остальные параметры и требования к использованию этой функции аналогичны описанию функции *CWnd::OnHScroll*.

Помимо собственно прокрутки (которая может и не производиться) приложения могут обрабатывать код сообщения SB_THUMBTRACK для выполнения каких-либо дополнительных действий. Например Microsoft Word версии 7.0 при перемещении бегунка вертикальной полосы прокрутки выводит номер страницы, а в версии 8.0 дополнительно к номеру страницы также выводится текст текущего (для этой страницы) заголовка.

Входящий в состав библиотеки MFC класс *CScrollBar* позволяет создать полосу прокрутки и имеет в своем составе необходимые функции для управления полосой прокрутки.

Создание объекта

Создание объекта *CScrollBar* и собственно полосы прокрутки Windows в коде программы требует создания (вызова) конструктора класса и функции *Create*:

```
BOOL CScrollBar::Create(
   DWORD dwStyle,
   const RECT& rect,
   CWnd* pParentWnd,
   UINT nID)
```

Параметр *dwStyle* задает стиль полосы прокрутки, который может содержать комбинацию следующих значений:

SBS_BOTTOMALIGN	используется со стилем SBS_HORZ; нижний край полосы прокрутки выравнивается по нижнему краю прямоугольника, указанного в параметре *rect*
SBS_HORZ	назначает горизонтальную полосу прокрутки; если не задан ни один из стилей SBS_BOTTOMALIGN или SBS_TOPALIGN, то полоса прокрутки имеет высоту, ширину и расположение, передаваемые в параметре *rect*

SBS_LEFTALIGN	используется со стилем SBS_VERT; правый край полосы прокрутки выравнивается по правому краю прямоугольника, заданного в параметре *rect*; полоса прокрутки имеет ширину, заданную по умолчанию для системных полос прокрутки
SBS_SIZEBOX	определяет размер полосы прокрутки; если не задан ни один из стилей SBS_SIZEBOXBOTTOMRIGHTALIGN или SBS_SIZEBOXTOPLEFTALIGN, то полоса прокрутки имеет ширину, высоту и расположение, заданные в параметре *rect*
SBS_SIZEBOXBOTTOMRIGHTALIGN	используется со стилем SBS_SIZEBOX; правый нижний угол выравнивается по нижнему правому углу прямоугольника, заданного в параметре *rect*
SBS_SIZEBOXTOPLEFTALIGN	используется со стилем SBS_SIZEBOX; верхний левый угол полосы прокрутки выравнивается по нижнему правому углу прямоугольника, заданного в параметре *rect*
SBS_TOPALIGN	используется со стилем SBS_HORZ; верхний край полосы прокрутки выравнивается по верхнему краю прямоугольника, заданного в параметре *rect;* высота полосы прокрутки равняется высоте полосы прокрутки, установленной в системе по умолчанию
SBS_VERT	определяет вертикальную полосу прокрутки; если не задан ни один из стилей SBS_RIGHTALIGN или SB_LEFTALIGN, то полоса прокрутки имеет высоту, ширину и расположение передаваемые в параметре *rect*

Кроме вышеперечисленных, следует использовать следующие стили:

WS_CHILD	следует указывать, всегда при создании полосы прокрутки
WS_VISIBLE	следует указывать, если полоса прокрутки должна быть изначально видимой
WS_DISABLED	следует указывать, если полоса прокрутки должна быть изначально недоступной
WS_TABSTOP	следует указывать, если полоса прокрутки должна входить в группу элементов управления, которым по очереди передается фокус ввода при нажатии клавиши табуляции

Параметр *rect* указывает расположение и размер полосы прокрутки. Может быть структурой *RECT* или объектом класса *CRect.* Параметр *pParentWnd* задает родительское окно (обычно объект *CDialog*), и *не может* быть равен NULL, а параметр *nID* задает идентификатор полосы прокрутки.

Функции для работы с полосой прокрутки

После вызова функции *Create*, а также после создания полосы прокрутки, заданной в качестве ресурсов, минимальное и максимальное значения границ прокрутки равны 0, также равен нулю и размер страницы (см. описание структуры *SCROLLINFO* далее в тексте).

Для того чтобы получить значения границ прокрутки, предназначена пара следующих функций:

void CScrollBar::**GetScrollRange**(
LPINT lpMinPos,
LPINT lpMaxPos) —

позволяет получить значения для границ прокрутки. Параметр lpMinPos — указатель на переменную целого типа, в которую записывается значение нижней границы полосы прокрутки, а параметр lpMaxPos — указатель на переменную целого типа, в которую записывается значение верхней границы полосы прокрутки.

Функция *SetScrollRange*, напротив, позволяет установить новые значения для границ прокрутки:

void CScrollBar::**SetScrollRange**(
int *nMinPos*,
int nMaxPos,
BOOL bRedraw = TRUE)

где параметр *bRedraw* задает, должна ли полоса прокрутки быть перерисована для отражения произведенных изменений. Если *bRedraw* = TRUE, то полоса прокрутки должна быть перерисована, если *bRedraw* = FALSE — то нет. Установка обеих границ равными нулю приводит к тому, что полоса прокрутки окна, в соответствии с заданным стилем, становится невидимой.

Если за вызовом функции *SetScrollRange* следует вызов функции *SetScrollPos*, то параметр *bRedraw* нужно установить в FALSE во избежание перерисовки полосы прокрутки дважды. Разница значений между параметрами, задаваемыми в *nMinPos* и *nMaxPos* не должна превышать 32767.

Для того чтобы определить или установить текущее положение бегунка на полосе прокрутки, предназначены следующие функции.

int CScrollBar::**GetScrollPos**() —

возвращает текущее положение бегунка полосы прокрутки. (В случае ошибки возвращается 0).

Текущее положение бегунка полосы прокрутки представляет собой относительное значение, которое зависит от текущего диапазона прокрутки. Например, если значения для прокрутки лежат в диапазоне от 100 до 200 и бегунок полосы прокрутки находится в ее середине, то текущая позиция будет равна 150.

int CScrollBar::**SetScrollPos**(
int nPos,
BOOL bRedraw = TRUE) —

устанавливает текущее положение бегунка на полосе прокрутки, задаваемое параметром nPos, которое должно лежать в заданном диапазоне, и при bRedraw = TRUE производит перерисовку полосы прокрутки для отражения нового положения.

```
...
// задаем пределы прокрутки от 50 до 150,
// и во избежание повторной перерисовки
// задаем параметр bRedraw = FALSE
m_ScrollBar.SetScrollRange(50, 150, FALSE);
// устанавливаем бегунок в среднее положение
// на полосе прокрутки и производим ее перерисовку
m_ScrollBar.SetScrollPos(100, TRUE)
...
```

Если необходимо сделать полосу прокрутки видимой или невидимой, то следует воспользоваться следующей функцией:

void CScrollBar::**ShowScrollBar**(BOOL bShow = TRUE)

которая показывает или скрывает полосу прокрутки, причем приложение не должно вызывать эту функцию для скрытия полосы прокрутки в момент обработки сообщения от нее.

Для того чтобы сделать полосу прокрутки полностью или частично недоступной, предназначена следующая функция:

BOOL CScrollBar::**EnableScrollBar**(UINT nArrowFlags = ESB _ENABLE_BOTH)

которая делает доступными или недоступными одну или обе кнопки со стрелками на полосе прокрутки. Функция возвращает значение TRUE, если произведены запрашиваемые изменения и FALSE — если стрелки уже находятся в запрашиваемом состоянии или произошла ошибка. Параметр nArrowFlags задает, какие кнопки со стрелками должны быть сделаны доступными или недоступными. Параметр может принимать следующие значения:

ESB_ENABLE_BOTH	сделать доступными обе кнопки со стрелками на полосе прокрутки
ESB_DISABLE_LTUP	сделать недоступными левую кнопку со стрелкой на горизонтальной или верхнюю на вертикальной полосе прокрутки
ESB_DISABLE_RTDN	сделать недоступными правую кнопку со стрелкой на горизонтальной или нижнюю на вертикальной полосе прокрутки
ESB_DISABLE_BOTH	сделать недоступными обе кнопки со стрелками на полосе прокрутки

Флаги имеют значения, при которых запрещение одной стрелки ведет к *автоматическому* разрешению другой.

Для получения или изменения параметров полосы прокрутки предназначена следующая пара функций:

BOOL CScrollBar::**GetScrollInfo**(
  LPSCROLLINFO lpScrollInfo,
  UINT nMask) —
    возвращает структуру, содержащую параметры полосы прокрутки.

BOOL CScrollBar::**SetScrollInfo**(
  LPSCROLLINFO lpScrollInfo,
  BOOL bRedraw = TRUE) —
    устанавливает параметры полосы прокрутки.

Параметр *lpScrollInfo* в обоих функциях является указателем на структуру *SCROLLINFO*, а параметр *nMask* первой функции указывает, какие параметры полосы прокрутки должны быть переданы. Обычно используются в качестве маски флаги SIF_ALL и комбинации из SIF_PAGE, SIF_POS, SIF_TRACKPOS и SIF_RANGE (см. ниже описание структуры *SCROLLINFO*).

Параметр *bRedraw* позволяет управлять режимом перерисовки полосы прокрутки в случае изменения ее свойств.

Структура *SCROLLINFO* содержит информацию о полосе прокрутки, включая минимальное и максимальное значения границ прокрутки, размер страницы и положение бегунка на полосе прокрутки. Она имеет следующее описание:

```
typedef struct tagSCROLLINFO {
  UINT cbSize;
  UINT fMask;
  int nMin;
  int nMax;
  UINT nPage;
  int nPos;
  int nTrackPos;
} SCROLLINFO;
typedef SCROLLINFO, FAR *LPSCROLLINFO;
```

cbSize — задает размер структуры (в байтах); *fMask* — задает параметры полосы прокрутки и может быть комбинацией следующих значений:

SIF_ALL	комбинация значений SIF_PAGE, SIF_POS и SIF_RANGE
SIF_DISABLENOSCROLL	используется только при установке параметров полосы прокрутки; если новые параметры полосы прокрутки делают ее ненужной, то это значение приводит к тому, что полоса прокрутки становится недоступной вместо того, чтобы стать невидимой
SIF_PAGE	параметр *nPage* содержит размер страницы

SIF_POS — параметр *nPos* содержит положение бегунка на полосе прокрутки

SIF_RANGE — параметры *nMin* и *nMax* содержат информацию о минимальном и максимальном значениях границ прокрутки

nMin — задает минимальное значение границы для прокрутки; *nMax* — задает максимальное значение границы для прокрутки; *nPage* — задает размер страницы для автоматического определения текущего положения бегунка при прокрутке на одну страницу (по умолчанию размер страницы равен 0); *nPos* — задает положение бегунка на полосе прокрутки; *nTrackPos* — задает непосредственное положение бегунка на полосе прокрутки в момент его перемещения при помощи мыши. Приложение не может изменять этот параметр, и функция *SetScrollInfo* игнорирует это поле.

При запросе параметров полосы прокрутки структуру *SCROLLINFO нет необходимости* инициализировать какими-либо значениями, включая поле, определяющее размер структуры, и поле маски.

Следующая функция избавляет от необходимости создавать дополнительные переменные для определения границ прокрутки, в случае если минимальное значение границы равно 0. Для этого можно воспользоваться вызовом функции *GetScrollLimit*.

int CScrollBar::**GetScrollLimit**() —

возвращает максимальный предел для прокрутки, в случае ошибки возвращает 0. К сожалению, эта функция возвращает значение *большее на единицу* задаваемого при вызове функции CScrollBar::SetScrollRange.

Для полос прокрутки, созданных в окнах в соответствии с заданным стилем окон, нет возможности получения доступа к объектам класса *CScrollBar* (см. описание функции *CWnd::OnHScroll* в начале раздела). Однако сам класс *CWnd* имеет в своем составе функции, аналогичные по назначению и наименованию функциям класса *CScrollBar*.

Глава 22

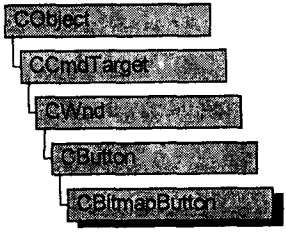
Классы с расширенными возможностями

Библиотека классов предоставляет разработчикам три дополнительных класса с расширенными возможностями, производных от классов, инкапсулирующих возможности стандартных элементов управления Windows.

Класс	Базовый класс	Элемент управления
CBitmapButton	*CButton*	кнопки с заданными рисунками для всех возможных состояний
CCheckListBox	*CListBox*	список элементов, реализованных в виде флажков
CDragListBox	*CListBox*	список с возможностью переноса его элементов при помощи мыши

Далее мы подробно остановимся на каждом из этих трех классов.

Кнопки с нанесенным битовым изображением (класс CBitmapButton)



Класс *CBitmapButton* предназначен для создания кнопок, на которых вместо текста будет размещено битовое изображение. Кнопка может иметь до четырех растровых изображений, вывод которых в зависимости от состояния кнопки и поддерживает класс *CBitmapButton*. Суть реализации этого класса в том, что он позволяет создавать командные кнопки — объекты Windows, отображение которых берет на себя приложение. В данном случае, отображением кнопки во всевозможных состояниях занимаются функции класса, использующие созданные разработчиком изображения. Состояния, определенные для кнопок:

- нормальное (доступное, ненажатое);
- кнопка нажата;
- кнопка имеет фокус ввода;
- кнопка недоступна.

В принципе, достаточно определить только один битовый массив для первого состояния, чтобы создать объект рассматриваемого класса. Рамка кнопки должна содержаться в самом изображении. Размер битовых массивов может быть любым, но размер собственно объекта-кнопки определяется первым заданным битовым массивом, т. е. массивом для нормального состояния кнопки. Если битовые массивы для других состояний имеют размеры, отличные от размера ненажатой кнопки, следует привести размер кнопки в соответствие с размером конкретного битового массива.

Различные виды применения требуют различное количество изображений для кнопки:

Состояние кнопки				
нормальное	**нажата**	**имеет фокус ввода**	**недоступна**	**Применение**
X				Вывод битового массива
X	X			Кнопка без стиля WS_TABSTOP
X	X	X	X	Кнопка со всеми состояниями в блоке диалога
X	X	X		Кнопка со стилем WS_TABSTOP в блоке диалога
				

При создании кнопки в шаблоне блока диалога необходимо обязательно установить стиль BS_OWNERDRAW. При создании кнопки чисто программным путем, все, что требуется, сделают функции класса *CBitmapButton*.

Кнопки в блоке диалога

Для включения кнопки в блок диалога необходимо выполнить следующие шаги:

1. Создать от одного до четырех битовых массивов (по мере необходимости), которые будут отображать состояния кнопки.

2. Создать шаблон блока диалога, определив в нем самоотображающуюся (owner redraw) кнопку. Размер кнопки не имеет значения, так как при размещении ее размер будет изменен в соответствии с размером битового массива.

> **Примечание**
>
> Обратите внимание на то, что поиск и загрузка битовых массивов для кнопки осуществляется на основе строки, которая задана для размещения на кнопке (хотя в действительности она на кнопку помещена не будет, поиск битовых изображений осуществляется с ее помощью). Поэтому следует задать строку текста для помещения на кнопке (например, как "IMAGE") и идентификатор кнопки (IDC_IMAGE). Для каждого битового изображения необходимо задать строго определенное имя: оно образуется путем добавления к имени, использованному при создании кнопки (в нашем примере — "IMAGE"), букв "U", "D", "F" или "X" ("U" — для нормального состояния кнопки (up), "D" — для нажатого (down), "F" — для кнопки, имеющей фокус ввода (focused), и "X" — когда кнопка недоступна). В данном случае битовые изображения получат идентификаторы "IMAGEU", "IMAGED", "IMAGEF" и "IMAGEX".
> Следует еще раз особо подчеркнуть, что битовые изображения имеют не числовые, а *строковые* идентификаторы. В противном случае битовые изображения не будут найдены и, соответственно, отображены.

3. В класс, производный от *CDialog*, необходимо добавить объект *CBitmapButton*.
4. Из функции *CDialog::OnInitDialog* вызвать функцию *CBitmapButton::AutoLoad*, указав в качестве параметров идентификатор кнопки, в нашем случае это IDC_IMAGE, и указатель на родительское окно — *this*.

Кнопки, созданные программным путем

При создании кнопки внутри окна необходимо выполнить следующие шаги:

1. Создать от одного до четырех битовых массивов (по мере необходимости), которые будут отображать состояния кнопки.
2. Создать объект класса *CBitmapButton*.
3. Вызвать функцию *Create* для создания элемента управления Windows "кнопка" и связи его с объектом класса *CBitmapButton*.
4. Вызвать функцию *CBitmapButton::LoadBitmaps* для загрузки ресурсов (битовых массивов) после того, как кнопка уже создана.

Функции-члены класса

Сообщения, посылаемые объектом класса *CBitmapButton* имеют тот же смысл и обрабатываются так же, как и при работе с обычной кнопкой.

Ниже приведем описание функций класса *CBitmapButton*.

Как обычно, конструктор класса создает лишь объект *CBitmapButton*, никак не воздействуя на Windows. Программное создание объекта Windows, как обычно, требует вызова функции *Create*. Но в отличие от ранее описанных классов, *CBitmapButton* не имеет в своем составе такой функции — создание объекта будет осуществляться при помощи функции базового класса, т. е. *CButton::Create*. Это скорее недоработка, чем попытка дать возможность известной гибкости разработчикам приложений. Поэтому важно при таком создании рассматриваемых кнопок не забыть указать стиль BS_OWNERREDRAW.

Для загрузки битовых изображений предназначены функции:

```
BOOL CBitmapButton::LoadBitmaps(
  LPCTSTR lpszBitmapResource,
  LPCTSTR lpszBitmapResourceSel = NULL,
  LPCTSTR lpszBitmapResourceFocus = NULL,
  LPCTSTR lpszBitmapResourceDisabled = NULL)
```

и

```
BOOL CBitmapButton::LoadBitmaps(
  UINT nIDBitmapResource,
  UINT nIDBitmapResourceSel = 0,
  UINT nIDBitmapResourceFocus = 0,
  UINT nIDBitmapResourceDisabled = 0)
```

которые позволяют загружать ресурсы, имеющие как строковые идентификаторы, так и числовые.

Данную функцию можно использовать, если нужно задать битовые массивы для помещения на кнопку по их именам или идентификаторам (для каждого индивидуально) или тогда, когда невозможно использование функции *CBitmapButton::AutoLoad*, например, при создании кнопки вне блока диалога.

При создании кнопки в блоке диалога целесообразнее использовать функцию

```
BOOL CBitmapButton::AutoLoad(
  UINT nID,
  CWnd* pParent)
```

Параметр nID задает идентификатор кнопки, а параметр pParent является указателем на объект-владелец кнопки, т. е. блок диалога.

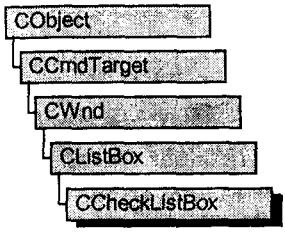
При использовании данной функции следует строго соблюдать порядок создания кнопки и битовых массивов, предназначенных для размещения на ней (см. раздел "Кнопки в блоке диалога").

Следующая функция позволяет изменять размер кнопки в соответствии с размером уже загруженных битовых массивов:

```
void CBitmapButton::SizeToContent()
```

Эту функцию следует использовать в случае, если битовые массивы имеют различные размеры, и только тогда, когда происходит смена состояний кнопки.

Список, имеющий кнопки выбора (класс CCheckListBox)



Класс *CCheckListBox* позволяет создать элемент управления, появившийся в Windows 95, сочетающий свойства списка и флажков (кнопок выбора): он представляет собой обычный список (LIST BOX), в котором к каждой строке добавлен слева флажок (CHECK BOX), устанавливая или снимая который пользователь может выбирать строки (элементы) списка.

Создаваемый таким образом список всегда является списком, внешний вид которого определяется пользователем, так как каждый его элемент содержит не только текстовую строку. В простейшем случае список может содержать текстовые строки и флажки в каждой строке, но это совсем не обязательно, можно определить список, в котором будут выводиться флажки и небольшие растровые изображения.

Если создается список, содержащий только текст и флажки, то для вывода строк списка можно использовать функцию по умолчанию *CCheckListBox::DrawItem*. Но в любом случае необходимо переопределить функции *CListBox::CompareItem*, *CCheckListBox::DrawItem* и *CCheckListBox::MeasureItem*.

Инициализация

Создание объекта *CCheckListBox* и собственно списка Windows в коде программы требует задания (вызова) конструктора класса и функции *Create*:

```
BOOL CCheckListBox::Create(
  DWORD dwStyle,
  const RECT & rect,
  CWnd* pParentWnd,
  UINT nID) —
```

создает элемент управления Windows и связывает его с объектом класса CCheckListBox. Параметр dwStyle задает стиль списка. Параметр должен содержать стиль LBS_OWNERDRAWFIXED (все элементы в списке одинаковой высоты) или LBS_OWNERDRAWVARIABLE (элементы в списке имеют разную высоту). Эти стили можно комбинировать со стилями обычного списка (см. описание класса CListBox). Параметр rect задает размер и расположение списка на экране, параметр pParentWnd задает родительское окно (обычно объект CDialog), который *не может* быть равен NULL, параметр nID задает идентификатор списка.

Функции для работы со списком

Следующая пара функций позволяет определить и установить состояние флажка для элемента списка:

int CCheckListBox::**GetCheck**(int nIndex) —
возвращает состояние флажка для элемента списка.

void CCheckListBox::**SetCheck**(
int nIndex,
int nCheck) —
устанавливает состояние флажка.

Параметр *nIndex* является номером элемента в списке, а параметр *nCheck* определяет состояние флажка: 0 — сброшен, 1 — установлен, 2 — неопределенное состояние.

Для того чтобы сделать доступным/недоступным элемент списка и определить его состояние, предназначены следующие функции:

BOOL CCheckListBox::**IsEnabled**(int nIndex) —
определяет, доступен ли элемент списка

void CCheckListBox::**Enable**(
int nIndex,
BOOL bEnabled = TRUE) —
делает доступным или недоступным элемент списка.

Для получения и установления стиля флажков списка предназначены функции:

UINT CCheckListBox::**GetCheckStyle**() —
возвращает стиль флажков.

void CCheckListBox::**SetCheckStyle**(UINT nStyle)
Параметр nStyle задает стиль флажков. Флажки могут иметь следующие стили: BS_CHECKBOX, BS_AUTOCHECKBOX, BS_AUTO3STATE или BS_3STATE (см. описание класса *CButton*).

Виртуальные функции

virtual CRect CCheckListBox::**OnGetCheckPosition**(
CRect rectItem,
CRect rectCheckBox) —
вызывается библиотекой для определения положения флажка записи списка, где rectItem определяет расположение и размер элемента списка, а rectCheckBox — размер и расположение (по умолчанию) флажка строки списка.

Реализация по умолчанию этой функции возвращает размер и расположение флажка, созданного с параметрами по умолчанию (выровнен по левому верхнему углу записи и имеет стандартный размер). Однако флажок может располагаться в правой части записи списка и иметь размер больше или меньше размера, установленного по умолчанию. В этих случаях необходимо переопределить эту функцию для того, чтобы библиотека могла получить правильные значения.

В следующем примере функция переопределяет функцию по умолчанию, помещает флажок в правой части элемента списка и переопределяет его размер:

```
CRect CMercuryCheckListBox::OnGetCheckPosition(
  CRect rectItem, CRect rectCheckBox)
{
  CRect rectMyCheckBox;
  rectMyCheckBox.top = rectItem.top -1;
  rectMyCheckBox.bottom = rectItem.bottom -1;
  rectMyCheckBox.right = rectItem.right -1;
  rectMyCheckBox.left =
    rectItem.right -1 - rectCheckBox.Width();

  return rectMyCheckBox;
}
```

virtual void CCheckListBox::**DrawItem**(LPDRAWITEMSTRUCT lpDrawItemStruct) —
вызывается библиотекой при необходимости перерисовки элемента списка, внешний вид которого определяется пользователем (см. описание функции CListBox::DrawItem).

virtual void CCheckListBox::**MeasureItem**(
LPMEASUREITEMSTRUCT lpMeasureItemStruct) —
вызывается библиотекой при необходимости определения высоты элемента списка, внешний вид которого определяется пользователем (см. описание функции CListBox::MeasureItem).

Следующий пример иллюстрирует создание списка, содержащего флажки, с использованием класса *CChecklistBox*.

```
/*************************************************************
Файл CheckBoxDlg.h
Copyright (c) 1997 Мешков А., Тихомиров Ю., Шведов Д.
*************************************************************/

// CheckBoxDlg.h : header file
//

#include "ColorBox.h"

/////////////////////////////////////////////////////////////
// CCheckBoxDlg dialog

class CCheckBoxDlg : public CDialog
{
// Construction
public:
  CCheckBoxDlg(CWnd* pParent = NULL);  // standard constructor

// Dialog Data
  //{{AFX_DATA(CCheckBoxDlg)
```

```
    enum { IDD = IDD_CHECKBOX_DIALOG };
    // NOTE: the ClassWizard will add data members here
  //}}AFX_DATA

  // ClassWizard generated virtual function overrides
  //{{AFX_VIRTUAL(CCheckBoxDlg)
  protected:
  virtual void DoDataExchange(CDataExchange* pDX);
  //}}AFX_VIRTUAL

// Implementation
protected:
  HICON m_hIcon;
  CColorBox m_CheckBox;

  // Generated message map functions
  //{{AFX_MSG(CCheckBoxDlg)
  virtual BOOL OnInitDialog();
  afx_msg void OnPaint();
  afx_msg HCURSOR OnQueryDragIcon();
  //}}AFX_MSG
  DECLARE_MESSAGE_MAP()
};

/*****************************************************************
Файл CheckBoxDlg.cpp
Copyright (c) 1997 Мешков А., Тихомиров Ю., Шведов Д.
*****************************************************************/

// CheckBoxDlg.cpp : implementation file
//

#include "stdafx.h"
#include "CheckBox.h"
#include "CheckBoxDlg.h"

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

/////////////////////////////////////////////////////////////
// CCheckBoxDlg dialog

CCheckBoxDlg::CCheckBoxDlg(CWnd* pParent /*=NULL*/)
  : CDialog(CCheckBoxDlg::IDD, pParent)
{
  //{{AFX_DATA_INIT(CCheckBoxDlg)
    // NOTE: the ClassWizard will add member initialization here
  //}}AFX_DATA_INIT
```

```
  // Note that LoadIcon does not require a subsequent
  // DestroyIcon in Win32
  m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
}

void CCheckBoxDlg::DoDataExchange(CDataExchange* pDX)
{
  CDialog::DoDataExchange(pDX);
  //{{AFX_DATA_MAP(CCheckBoxDlg)
    // NOTE: the ClassWizard will add DDX and DDV calls here
  //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CCheckBoxDlg, CDialog)
  //{{AFX_MSG_MAP(CCheckBoxDlg)
  ON_WM_PAINT()
  ON_WM_QUERYDRAGICON()
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()

////////////////////////////////////////////////////////
// CCheckBoxDlg message handlers

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

  // Set the icon for this dialog.
  // The framework does this automatically
  //   when the application's main window is not a dialog
  SetIcon(m_hIcon, TRUE);      // Set big icon
  SetIcon(m_hIcon, FALSE);     // Set small icon

  // связываем объект класса CColorBox с элементом
  // управления по идентификатору
  VERIFY(m_CheckBox.SubclassDlgItem(IDC_CHECKBOX, this));

  // заполняем список строками,
  // в которых выводятся различные цвета -
  // цветов всего 8, и их определяет наличие или
  // отсутствие сооставляющей
  for (int red = 0; red <= 255; red += 255)
    for (int green = 0; green <= 255; green += 255)
      for (int blue = 0; blue <= 255; blue += 255)
        m_CheckBox.AddColorItem(RGB(red, green, blue));

  return TRUE;  // return TRUE  unless you set
                // the focus to a control
}

...
```

```
/************************************************************
Файл ColorBox.h
Copyright (c) 1997 Мешков А., Тихомиров Ю., Шведов Д.
************************************************************/

// ColorBox.h : header file
//

/////////////////////////////////////////////////////////
// CColorBox window

class CColorBox : public CCheckListBox
{
// Construction
public:
  CColorBox();

// Attributes
public:

// Operations
public:
  void AddColorItem(COLORREF color);

// Overrides
  // ClassWizard generated virtual function overrides
  //{{AFX_VIRTUAL(CColorBox)
  public:
  virtual int CompareItem(
    LPCOMPAREITEMSTRUCT lpCompareItemStruct);
  virtual void DrawItem(
    LPDRAWITEMSTRUCT lpDrawItemStruct);
  virtual void MeasureItem(
    LPMEASUREITEMSTRUCT lpMeasureItemStruct);
  //}}AFX_VIRTUAL

// Implementation
public:
  virtual ~CColorBox();

  // Generated message map functions
protected:
  //{{AFX_MSG(CColorBox)
    // NOTE - the ClassWizard will add and
    //        remove member functions here.
  //}}AFX_MSG

  DECLARE_MESSAGE_MAP()
};

/////////////////////////////////////////////////////////
```

```
/*************************************************************
Файл ColorBox.cpp
Copyright (c) 1997 Мешков А., Тихомиров Ю., Шведов Д.
*************************************************************/

// ColorBox.cpp : implementation file
//

#include "stdafx.h"
#include "CheckBox.h"
#include "ColorBox.h"

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

#define COLOR_ITEM_HEIGHT 20

/////////////////////////////////////////////////////////
// CColorBox

CColorBox::CColorBox()
{
}

CColorBox::~CColorBox()
{
}

BEGIN_MESSAGE_MAP(CColorBox, CCheckListBox)
  //{{AFX_MSG_MAP(CColorBox)
    // NOTE - the ClassWizard will add and
    //        remove mapping macros here.
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()

void CColorBox::AddColorItem(COLORREF color)
{
  // добавляем строку с список
  AddString((LPCTSTR) color);
}

/////////////////////////////////////////////////////////
// CColorBox message handlers

// переопределяем функцию для сравнения строк списка
int CColorBox::CompareItem(
  LPCOMPAREITEMSTRUCT lpCompareItemStruct)
{
  // return -1 = item 1 sorts before item 2
```

```
  // return 0 = item 1 and item 2 sort the same
  // return 1 = item 1 sorts after item 2

  COLORREF cr1 = (COLORREF)lpCompareItemStruct->itemData1;
  COLORREF cr2 = (COLORREF)lpCompareItemStruct->itemData2;

  if (cr1 == cr2)
   return 0; // строки одинаковы

  // сначала сортирем строки по интенсивности
  int intensity1 = GetRValue(cr1) +
                   GetGValue(cr1) +
                   GetBValue(cr1);

  int intensity2 = GetRValue(cr2) +
                   GetGValue(cr2) +
                   GetBValue(cr2);

  if (intensity1 < intensity2)
    return -1; // сначала идут строки с меньшей яркостью
  else if (intensity1 > intensity2)
    return 1; // более яркие строки идут вторыми

  // если строки имеют одинаковую интенсивность,
  // то сортируем по цветам:
  //   первый - синий, последний - красный
  if (GetBValue(cr1) > GetBValue(cr2))
    return -1;
  else if (GetGValue(cr1) > GetGValue(cr2))
    return -1;
  else if (GetRValue(cr1) > GetRValue(cr2))
    return -1;
  else
    return 1;

  return 0;
}

// переопределяем функцию для рисования строк списка,
// причем рисуем только содержимое каждого из
// элементов - квадрат и знак в нем (установлен/сброшен)
// рисуются без нашего участия
void CColorBox::DrawItem(
  LPDRAWITEMSTRUCT lpDrawItemStruct)
{
  CDC* pDC = CDC::FromHandle(lpDrawItemStruct->hDC);
  COLORREF cr = (COLORREF)lpDrawItemStruct->itemData;

  if (lpDrawItemStruct->itemAction & ODA_DRAWENTIRE)
  {
    CBrush br(cr);
```

```
    pDC->FillRect(&lpDrawItemStruct->rcItem, &br);
  }

  if ((lpDrawItemStruct->itemState & ODS_SELECTED) &&
    (lpDrawItemStruct->itemAction &
    (ODA_SELECT | ODA_DRAWENTIRE)))
  {
    // выделение рамки, если запись выделена
    COLORREF crHilite = RGB(255-GetRValue(cr),
                            255-GetGValue(cr),
                            255-GetBValue(cr));

    CBrush br(crHilite);
    pDC->FrameRect(&lpDrawItemStruct->rcItem, &br);
  }

  if (!(lpDrawItemStruct->itemState & ODS_SELECTED) &&
    (lpDrawItemStruct->itemAction & ODA_SELECT))
  {
    // удаление рамки, если выделение с записи списка снято
    CBrush br(cr);
    pDC->FrameRect(&lpDrawItemStruct->rcItem, &br);
  }
}

// переопределяем функцию для определения размеров строк списка
void CColorBox::MeasureItem(
  LPMEASUREITEMSTRUCT lpMeasureItemStruct)
{
  lpMeasureItemStruct->itemHeight = COLOR_ITEM_HEIGHT;
}
```

После запуска приложения на экран будет выведен блок диалога (рис. 20), содержащий список, в котором строки будут иметь различные цвета:

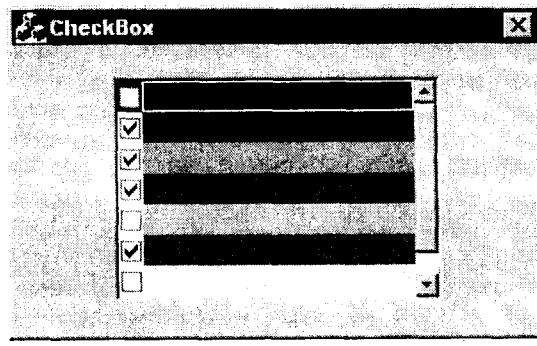
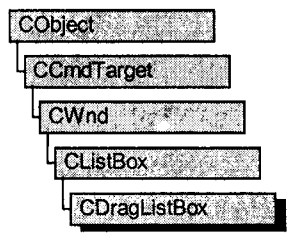


Рис. 20. Блок диалога уже после осуществления выбора элементов

Обратите внимание на то, что связывание члена CCheckBoxDlg::m_CheckBox происходит без механизма динамического обмена данными (DDX — Dynamic Data Exchange). Дело в том, что в версии Visual C++ 4.x средство ClassWizard не поддерживает класс *CCheckListBox*. (В примере класс *CColorBox* изначально был создан на основе класса *CListBox*, затем вручную имя базового класса было заменено на *CCheckListBox*).

Список с возможностью перемещения элементов (класс CDragListBox)



Класс *CDragListBox* позволяет создать список, который в дополнение к свойствам обычных списков позволяет пользователю перетаскивать элементы списка внутри него. Списки, обладающие этим свойством, позволяют пользователю упорядочивать элементы и могут оказаться в некоторых случаях очень удобными, например, для перемещения имен файлов в списке. По умолчанию, при перемещении элемента список помещает его в новое место, однако есть возможность задать параметры так, что элемент будет не перемещаться, а копироваться.

Ограничения на использование

По умолчанию список не может иметь стили LBS_MULTIPLESEL и LBS_SORT. Исключение последнего стиля, определяющего необходимость автоматической сортировки, вполне естественно. Множественный выбор в принципе может пригодиться, хотя тогда не совсем ясно, как реализовывать механизм перетаскивания или копирования, т. к. эти операции производятся только внутри последовательного списка. Кроме того, собственно операция перетаскивания может быть осуществлена только для списков, содержащих строки, т. е. объект-список должен иметь стиль LBS_HASSTRINGS или не иметь стили LBS_OWNERDRAWFIXED или LBS_OWNERDRAWVARIABLE. (Способ позволяющий отказаться от последнего из перечисленных ограничений изложен в конце раздела.)

Создание списка

Для создания объекта класса *CDragListBox* необходимо использовать конструктор. Для создания списка с возможностью перетаскивания элементов на основе шаблона блока диалога, следует создать обычный элемент управления "список" (LIST BOX) и связать с ним объект класса *CDragListBox* или производного от него. Естественно, что список не должен иметь стили, перечисленные выше.

Функции, предназначенные для перемещения элементов

Класс *CDragListBox* является производным от класса *CListBox* и обладает всеми его свойствами, но кроме того он имеет и свои собственные функции, обеспечивающие использование его новых возможностей.

int CDragListBox::**ItemFromPt**(
CPoint pt,
BOOL bAutoScroll = TRUE) —

возвращает индекс перетаскиваемого элемента. Параметр pt — объект класса CPoint, содержащий координаты точки внутри списка, его следует использовать для получения индекса элемента списка. Параметр bAutoScroll определяет, будет ли производиться автоматически прокрутка списка.

virtual void CDragListBox::**DrawInsert**(int nItem) —

функция, вызываемая библиотекой для прорисовки указателя вставки элемента до того, как сам элемент будет вставлен. Параметр nItem содержит индекс элемента, в который будет производиться вставка. Значение -1 сбрасывает указатель вставки. Переопределите эту функцию, если необходимо задать свой указатель вставки элемента. Функция использует член класса m_nLast, сохраняющий номер элемента списка, после которого находился указатель вставки.

Собственно рисованием указателей нового места для вставки или копирования занимается недокументированная функция класса — *DrawSingle*. Эта функция вызывается дважды — для предыдущего положения и для текущего положения указателя вставки. Связано это с тем, что указатель рисуется в режиме PATINVERT, когда двойной вывод по одному и тому же месту восстанавливает исходное изображение.

Вызов функции *BeginDrag* сообщает о начале операции перемещения элемента списка:

virtual BOOL CDragListBox::**BeginDrag**(CPoint pt) —

эта функция возвращает значение TRUE, если операция перемещения разрешена, параметр pt определяет точку внутри окна списка.

Функция вызывается библиотекой тогда, когда возможно началась операция перемещения строки, например, когда пользователь нажал левую кнопку мыши. Эту функцию стоит переопределить в том случае, если необходимо контролировать, что происходит в начале операции перемещения записи списка. Реализация по умолчанию данной функции захватывает мышь, и строка остается в режиме перетаскивания, пока пользователь не отпустит левую или нажмет правую кнопку мыши или клавишу <Esc.> В последних двух случаях операция перемещения отменяется, и происходит вызов функции *CancelDrag*:

virtual void CDragListBox::**CancelDrag**(CPoint pt) —

параметр pt определяет точку внутри окна списка.

Во время перетаскивания обработчик уведомлений вызывает функцию *Dragging*:

virtual UINT CDragListBox::**Dragging**(CPoint pt) —

параметр pt передает экранные координаты текущего положения указателя мыши, который может находиться и вне окна списка. Функция возвращает идентификатор изображаемого курсора: [1]

DL_COPYCURSOR	обозначает режим копирования элемента (курсор со знаком "+")
DL_MOVECURSOR	обозначает режим перемещения элемента (обычный курсор)
DL_STOPCURSOR	обозначает, что элемент не может быть перемещен в указанное место (курсор в виде перечеркнутой окружности)

Реализация по умолчанию функции возвращает DL_MOVECURSOR.

И, наконец, последняя из виртуальных функций — функция *Dropped*:

virtual void CDragListBox::**Dropped**(
int nSrcIndex,
CPoint pt) —

эта функция вызывается библиотекой, когда запись перемещена (скопирована) и отпущена. Параметр nSrcIndex задает исходные координаты перемещаемого элемента, параметр pt содержит новые координаты.

Реализация по умолчанию данной функции копирует элемент списка и его данные в новое место и удаляет затем исходный элемент. Если необходимо реализовать копирование элементов при перемещении, то следует переопределить данную функцию.

Методика обхода ограничений

При желании можно допустить работу со списками, не имеющими стиля LBS_HASSTRINGS при наличии стилей LBS_OWNERDRAWFIXED или LBS_OWNERDRAWVARIABLE.

Для этого нужно не только написать свою версию функции *DrawItem*, но и полностью переписать функцию *Dropped*, в которой следует отказаться лишь от использования функций *GetText* и *SetText*.

Реализация копирования элементов списка

Данный пример иллюстрирует реализацию списка, позволяющего копировать элементы списка в указанное место. Элементами списка являются имена файлов текущего каталога, включая имена подкаталогов. Обратите

[1] Вид курсора зависит от системных установок, в таблице приведены описания стандартного вида курсоров.

внимание, что на рисунке отображен результат предыдущего копирования строки, содержащей имя файла readme.txt.

```
/*************************************************************
Файл DragListDlg.cpp
Copyright (c) 1997 Мешков А., Тихомиров Ю., Шведов Д.
*************************************************************/

// DragListDlg.cpp : implementation file
//

...

// связываем член m_DragList с элементом управления
void CDragListDlg::DoDataExchange(CDataExchange* pDX)
{
  CDialog::DoDataExchange(pDX);
  //{{AFX_DATA_MAP(CDragListDlg)
  DDX_Control(pDX, IDC_DRAGLIST, m_DragList);
  //}}AFX_DATA_MAP
}

...

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

  ...

  // заполняем элемент управления
  // именами файлов текущего каталога
  m_DragList.Dir(0xFF, "*.*");

  ...
}

...

/*************************************************************
Файл DragBox.h
Copyright (c) 1997 Мешков А., Тихомиров Ю., Шведов Д.
*************************************************************/

// DragBox.h : header file
//

/////////////////////////////////////////////////////////////
// CDragBox window

class CDragBox : public CDragListBox
{
...

// Operations
public:
```

```
// Overrides
  // ClassWizard generated virtual function overrides
  //{{AFX_VIRTUAL(CDragBox)
  //}}AFX_VIRTUAL
  virtual void DrawInsert(int nItem);
  virtual UINT Dragging(CPoint pt);
  virtual void Dropped(int nSrcIndex, CPoint pt);

...
};

/*************************************************************
Файл DragBox.cpp
Copyright (c) 1997 Мешков А., Тихомиров Ю., Шведов Д.
*************************************************************/

////////////////////////////////////////////////////
// DragBox.cpp : implementation file
//

...

// пример реализации функции рисования
// указателя перемещения
void CDragBox::DrawInsert(int nItem)
{
  // проверяем было ли перемещение
  // в новое место
  if (m_nLast != nIndex)
  {
    DrawSingle(m_nLast);
    DrawSingle(nIndex);

    m_nLast = nIndex;
  }
}

// основную работу пусть делает функция
// базового класса - мы только заменим
// тип курсора
UINT CDragBox::Dragging(CPoint pt)
{
  if (CDragListBox::Dragging(pt) == DL_MOVECURSOR)
    return DL_COPYCURSOR;

  return DL_STOPCURSOR;
}

void CDragBox::Dropped(int nSrcIndex, CPoint pt)
{
  // проверяем допустимость стилей списка
  ASSERT(!(GetStyle() & LBS_OWNERDRAWFIXED |
```

```
    LBS_OWNERDRAWVARIABLE)) ||
    (GetStyle() & LBS_HASSTRINGS));

  // удаляем указатель перемещения
  DrawInsert(-1);

  // определяем новое место для вставки
  int nDestIndex = ItemFromPt(pt);

  // проверка необходимости вставки
  if (nSrcIndex == -1 || nDestIndex == -1)
     return;

  if (nDestIndex == nSrcIndex || nDestIndex == nSrcIndex+1)
    return;

  // получаем информацию об элементе
  CString str;
  DWORD dwData;
  GetText(nSrcIndex, str);
  dwData = GetItemData(nSrcIndex);

  // вот теперь можно текущий элемент удалить -
  // тогда функция будет производить перемещение

  // вставляем новый элемент и связываем с ним данные
  if (nSrcIndex < nDestIndex)
    nDestIndex--;
  nDestIndex = InsertString(nDestIndex, str);
  SetItemData(nDestIndex, dwData);

  // назначаем новый элемент текущим
  SetCurSel(nDestIndex);
}
```

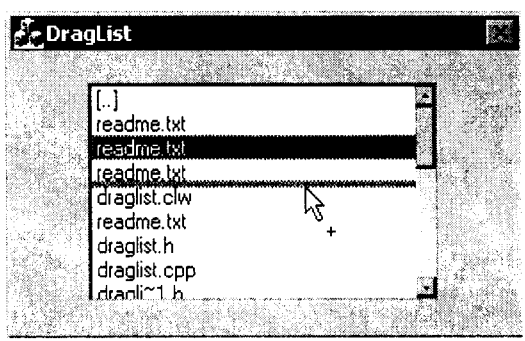


Рис. 21. Вид окна приложения во время выбора места для вставки копии имени файла

Часть IX

Общие элементы управления (Common Controls)

23. Использование общих элементов управления

24. Вспомогательные элементы управления

25. Комплексные элементы управления

Глава 23

Использование общих элементов управления

Обзор

Общие элементы управления появились как реакция на уже реализованные возможности во многих Windows-приложениях. Действительно, на сегодняшний день уже де-факто стало стандартом иметь в своих программных продуктах целый ряд небольших, но очень полезных элементов, позволяющих упростить ввод и обработку данных, отображать текущее состояние длительных во времени процессов и т. д. Естественно, что Windows позволяет реализовывать практически не ограниченное множество различных по возможностям элементов управления. Однако такие доморощенные средства не имеют должной поддержки со стороны операционной системы. Например, элементы управления, созданные на базе технологии OLE, так называемые элементы OCX (раньше — VBX), не получили широкого распространения прежде всего потому, что они являются в большинстве своем продуктами третьих фирм.

Общие элементы управления появились в следующих версиях Windows:

- Windows 95;
- Windows NT, начиная с версии 3.51;

Библиотека классов MFC обеспечивает поддержку всех новых общих элементов управления с помощью соответствующих классов. Некоторые элементы настолько просты в использовании, что создание соответствующего класса, инкапсулирующего его возможности, в принципе нужно только для поддержки самой идеологии библиотеки MFC. Поддержка с помощью классов других, более сложных элементов существенно облегчает их использование. Но в любом случае все элементы управления известны и средству генерации приложений — AppWizard и средству генерации классов — ClassWizard, поэтому независимо от сложности стандартного элемента управления Visual C++ всегда обеспечит его поддержку.

Несмотря на то, что именно эта часть книги посвящена общим элементам управления, в первом томе мы уже упоминали о них в главе 2 первого тома и подробно описывали с примерами назначение и использование двух общих элементов и соответствующих им классов (глава 13 первого тома):

- Строка состояния (классы *CStatusBar* и *CStatusBarCtrl*)
- Панель инструментов (классы *CToolBar* и *CToolBarCtrl*)

В последующих главах мы рассмотрим следующие общие элементы управления и соответствующие им классы:

- Просмотр видео-клипов (класс *CAnimateCtrl*)
- Клавиша вызова (класс *CHotKeyCtrl*)
- Индикатор (класс *CProgressCtrl*)
- Линейка с бегунком (класс *CSliderCtrl*)
- Спин (класс *CSpinButtonCtrl*)
- Подсказка (класс *CToolTipCtrl*)
- Список изображений (класс *CImageCtrl*)
- Заголовок (класс *CHeaderCtrl*)
- Просмотр списка (класс *CListCtrl*)
- Просмотр дерева (класс *CTreeCtrl*)

Общему элементу управления "расширенный редактор" (класс *CRichEditCtrl*) будет посвящена целая часть следующего тома нашей книги.

Общие элементы управления в блоках диалога

Как и все стандартные элементы управления, общие[1] могут быть использованы для блоков диалога, а также для любых других окон, строящихся на основе шаблонов блоков диалога — окон мастеров, свойств, форматных представлений и т. д.

Рассмотрим алгоритм использования общих элементов управления в блоках диалога, который может быть использован с небольшими изменениями и для форм.

1. После создания шаблона блока диалога, в него необходимо поместить в редакторе ресурсов необходимый элемент управления.
2. В класс, представляющий блок диалога, необходимо добавить переменную, связанную с элементом управления.

---

[1] За исключением строки состояния (класс *CStatusBarCtrl*), панели инструментов (класс *CToolBarCtrl*) и расширенного редактора (класс *CRichEditCtrl*).

3. В функции *OnInitDialog* класса, представляющего блок диалога, необходимо проинициализировать элемент управления.

Следует помнить, что переменные-члены диалогового класса связаны с элементом управления вплоть до завершения самого диалога, таким образом, нельзя допускать доступ к ним после завершения функции *CDialog::DoModal* или любой другой функции, активизирующей диалог. Для получения информации о последнем состоянии элемента управления (если это, конечно, необходимо), следует создать переменные-члены класса, не связанные с элементами управления, например, строковые или численные переменные, которые будут хранить последние значения, записанные в них, например, переопределенной функцией *OnOK*.

Общие элементы управления в качестве дочерних окон

Все общие элементы управления могут быть дочерними окнами для любых других окон. Рассмотрим алгоритм их динамического создания и работы с ними.

1. Создание объекта класса элемента управления.
2. Вызов функции *Create* соответствующего класса.
3. Создание собственно объекта Windows — общего элемента управления при помощи членов-функций соответствующего класса, после чего становится возможным управление и работа с объектом.
4. Вызов функции оконного класса *CWnd::DestroyWindow* по завершении использования элемента управления.

Уведомления от общих элементов управления

Все общие элементы управления являются дочерними окнами, которые посылают родительскому окну уведомления о самых разнообразных событиях. Естественно, что тип события определяется конкретным элементом управления.

Все сказанное выше в равной мере относится и к стандартным элементам управления, однако, в отличие от них, уведомления передаются не при помощи сообщения WM_COMMAND, а при помощи WM_NOTIFY.[1] Для обработки всех уведомляющих сообщений родительское окно, вернее, класс

[1] Как всегда из правила есть исключения, см. "Просмотр видео-клипов".

окна может переопределить виртуальные функции *CWnd::OnCommand* или *CWnd::OnNotify* соответственно. Для создания отдельных функций, обрабатывающих уведомления от общих элементов управления, можно воспользоваться макросом ON_NOTIFY.

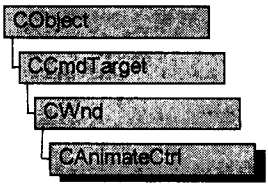
Такой способ позволяет избежать необходимости создания производных классов от классов элементов управления. Однако в ряде случаев создать производный класс все-таки бывает необходимо, например, для изменения цвета элемента управления, что возможно реализовать только при наличии производного класса. В этом случае все уведомляющие сообщения от дочернего окна родительскому будут как бы отражены (reflected) обратно дочернему.

Более подробно о создании функций обработки уведомляющих и отраженных уведомляющих сообщений рассказано в главе 10 первого тома книги.

Глава 24

Вспомогательные элементы управления

Просмотр видео-клипов (класс CAnimateCtrl)



Класс *CAnimateCtrl* обеспечивает функциональные возможности элемента управления Windows, который реализует просмотр видео-клипов (элемент анимации).

Windows поддерживает просмотр видеоинформации, записанной в формате AVI (Audio Video Interleaved) — стандартном формате Windows. Такие видео-клипы представляют собой совокупность растровых изображений — кадров. Сам просмотр осуществляется в прямоугольной области, размер которой определяется либо заданными размерами элемента управления, либо размерами изображения в зависимости от стиля элемента (см. ниже).

Для того чтобы просмотр мог быть осуществлен, видеоинформация должна соответствовать следующим требованиям (вообще, просмотр может быть осуществлен только простейших видео-клипов):

- клип должен иметь ровно один видеопоток, содержащий хотя бы один кадр;
- клип может иметь максимум два потока, один из которых обычно представляет звук и игнорируется;
- содержимое клипа может быть либо не сжато, либо сжато с использованием алгоритма RLE8;
- изменение палитры не допускается.[1]

---

[1] Для применения в приложениях более совершенных средств мультимедиа следует использовать окна класса *MCIWnd* (имеется в виду класс объектов Windows, а не класс C++).

Видео-клип может быть добавлен к приложению в качестве ресурса типа AVI, например, вот так:

```
IDR_VISUAL  AVI  DISCARDABLE  "res\\Visual.avi"
```

Иными словами, просмотр любых видеофайлов (соответствующих требованиям) может быть осуществлен по имени.

Применение элемента анимации

Microsoft рекомендует использовать этот элемент управления для некоторой визуализации длительных операций. Сама система использует, прежде всего, видео-клипы для визуализации копирования, перемещения и удаления файлов и папок, а также при поиске объектов, например, компьютеров, по сети.[1]

Разработчики программного обеспечения идут дальше. Система Remove-IT 95[2] компании Vertisoft Systems, Inc. широко использует анимацию в диалоге выбора команд, а также для визуализации работы монитора, собирающего информацию и т. д.

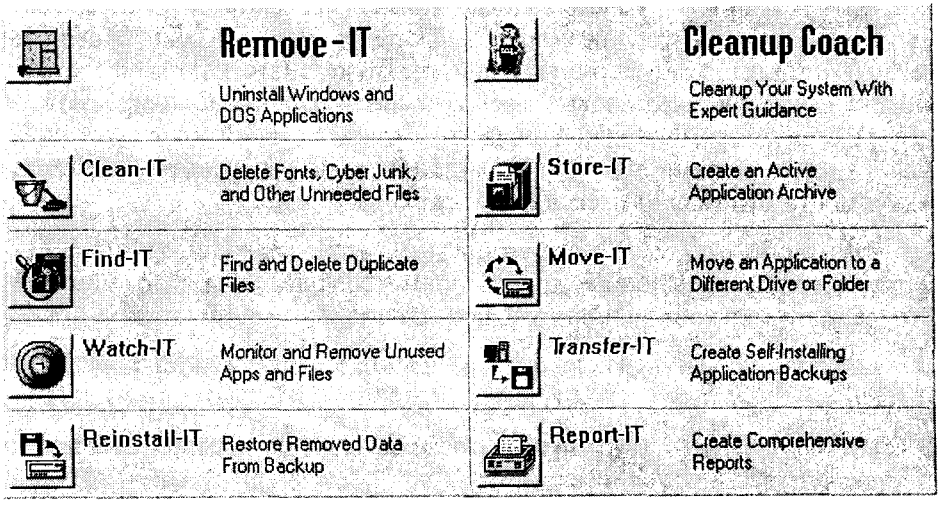


Рис. 22. Команды и соответствующие им кнопки с видео-клипами

[1] Приложение Internet Explorer, также корпорации Windows, использует анимацию в качестве элемента дизайна.

[2] Мониторинг использования различных компонентов системы, в основном приложений и отдельных файлов.

Использование класса CAnimateCtrl

Как и для других элементов управления, для просмотра видео-клипов достаточно или создать соответствующий объект в качестве дочернего в любом окне или добавить этот элемент управления в шаблон блока диалога. И тот и другой способ позволяет задавать дополнительные стили, свойственные именно для просмотра видео-клипов.

Конструктор создает только объект класса. Чтобы создать непосредственно объект Windows без использования шаблона блока диалога необходимо воспользоваться функцией *Create*:

```
BOOL CAnimateCtrl::Create(
  DWORD dwStyle,
  const RECT & rect,
  CWnd* pParentWnd,
  UINT nID)
```

Назначение параметров очевидно. Стоит только отметить, что параметр *dwStyle* может принимать любые комбинации стилей элементов управления плюс следующие, свойственные для рассматриваемого элемента управления:

ACS_CENTER	обеспечивает центрирование видео-клипа в рамках, заданных при создании элемента управления; при этом возможно исключение областей клипа, выходящих за рамки элемента управления; в противном случае размер элемента просмотра будет подстроен под размеры клипа; при этом левый верхний угол кадров клипа совпадет с верхним левым углом элемента управления
ACS_TRANSPARENT	обеспечивает прозрачный режим отображения видео-клипа, т. е. произойдет замена цвета фона кадров цветом фона родительского окна элемента просмотра
ACS_AUTOPLAY	обеспечивает немедленное и автоматическое начало "проигрывания" видео-клипа в цикле, т. е. продолжение просмотра с начала после завершения

Собственно элемент управления (класс *CAnimateCtrl*) не предусматривает большого числа возможностей для манипулирования видео-клипом. Всего существует пять функций и два уведомляющих сообщения, которые можно обрабатывать.

После создания объекта необходимо загрузить из ресурсов или AVI-файла видео-клип. Для этого используется функция *Open*. Вызов этой функции с параметром, равным NULL, приведет к освобождению ресурса, что также может быть достигнуто вызовом функции *Close*. Важной особенностью функции *Open* является то, что, не начиная "проигрывания" клипа, тем не менее, она производит отображение его первого кадра.[1]

---

[1] Стиль ACS_AUTOPLAY снимает это ограничение — задание этого стиля приведет к немедленному началу "проигрывания".

Для начала "проигрывания" класс предусматривает использование функции *Play*:

BOOL CAnimateCtrl::**Play**(
  UINT nFrom,
  UINT nTo,
  UINT nRep)

где параметры *nFrom* и *nTo* определяют начальный и конечный кадры, считая с нулевого, для "проигрывания". Оба значения должны лежать в пределах от 0 и до 65535 включительно. Значение параметра *nTo*, равное -1, указывает, что "проигрывание" необходимо проводить до конца клипа, таким образом, для "проигрывания" клипа полностью нет необходимости знать количество составляющих его кадров. Третий параметр определяет число "проигрываний" клипа. Значение -1 задает бесконечное "проигрывание".

Для остановки "проигрывания" видео-клипа в любой момент времени необходимо вызвать функцию *Stop*.

Для перемещения к определенному кадру, но без инициализации "проигрывания" служит функция *Seek*:

BOOL CAnimateCtrl::**Seek**(
  UINT nTo)

Единственный параметр определяет номер кадра, начиная с нуля, который необходимо отобразить. Значение параметра равное -1 указывает на необходимость отображения последнего кадра клипа.

Уведомления

Описываемый элемент управления посылает два вида уведомлений. Причем, как исключение из правила, оба уведомления посылаются не в форме сообщения WM_NOTIFY, что обычно для общих элементов управления, а в форме сообщения WM_COMMAND.

Первое сообщение — ACN_START — посылается приложению, а, точнее, родительскому окну, в момент начала "проигрывания" видео-клипа.

Второе сообщение — ACN_STOP — посылается по окончании процесса "проигрывания".

Для того чтобы обработать эти сообщения, в рамках библиотеки классов MFC существует макрос ON_CONTROL, при помощи которого определяют элемент уведомления, тип уведомления и функцию его обработки.

Для наглядности продемонстрируем пример (приложение Animate), который создан на основе блока диалога. Последний содержит два элемента управления для просмотра одного и того же видео-клипа. Суть примера в том, что "проигрывание" осуществляется попеременно то одним элементом управления, то другим. Переключение осуществляется по завершении работы одного из элементов.

```
/*************************************************************
Файл AnimateDlg.cpp
Copyright (c) 1997 Мешков А., Тихомиров Ю., Шведов Д.
*************************************************************/

// AnimateDlg.cpp : implementation file
//
...
//
// связываем объекты класса CAnimateCtrl
// с элементами блока диалога
//
void CAnimateDlg::DoDataExchange(CDataExchange* pDX)
{
  CDialog::DoDataExchange(pDX);
  //{{AFX_DATA_MAP(CAnimateDlg)
  DDX_Control(pDX, IDC_ANIMATE_1, m_Animate_1);
  DDX_Control(pDX, IDC_ANIMATE_2, m_Animate_2);
  //}}AFX_DATA_MAP
}

//
// добавляем в карту сообщений функции-обработчики
// уведомлений. (Добавление осуществляется вручную,
// т. к. ClassWizard не поддерживает эти уведомления)
//
BEGIN_MESSAGE_MAP(CAnimateDlg, CDialog)
  //{{AFX_MSG_MAP(CAnimateDlg)
  ON_WM_PAINT()
  ON_WM_QUERYDRAGICON()
  //}}AFX_MSG_MAP
  ON_CONTROL(ACN_STOP, IDC_ANIMATE_1, AnimateStop_1)
  ON_CONTROL(ACN_STOP, IDC_ANIMATE_2, AnimateStop_2)
END_MESSAGE_MAP()
...

BOOL CAnimateDlg::OnInitDialog()
{
  CDialog::OnInitDialog();
...

  // "открываем" клип для первого элемента управления
  m_Animate_1.Open(IDR_VISUAL);
  // и немедленно начинаем его просмотр
  m_Animate_1.Play(0, (UINT)-1, 1);

  // "открываем" клип для второго элемента управления
  m_Animate_2.Open(IDR_VISUAL);
...
}
```

```
...
//
// функция обработки уведомления о завершении просмотра
// от первого элемента управления - начинает просмотр
// во втором элементе управления
//
void CAnimateDlg::AnimateStop_1()
{
  m_Animate_2.Play(0, (UINT)-1, 1);
}

//
// функция обработки уведомления о завершении просмотра
// от второго элемента управления - начинает просмотр
// в первом элементе управления
//
void CAnimateDlg::AnimateStop_2()
{
  m_Animate_1.Play(0, (UINT)-1, 1);
}
```

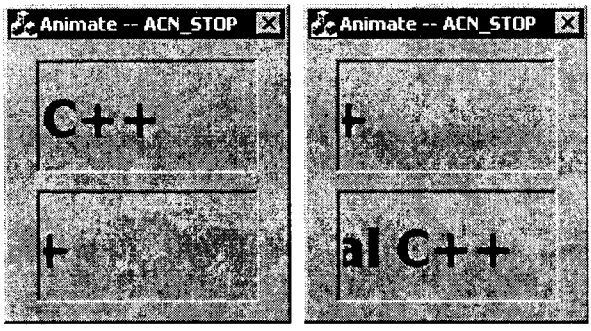


Рис. 23. Окно приложения Animate. Слева — активен верхний элемент управления, справа — активен нижний элемент управления

Создание видео-клипов

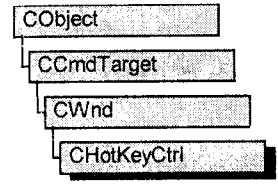
Итак, как просматривать видео-клипы мы научились. А как же все-таки их создавать? Для этого необходимо создать набор файлов формата BMP (кадров) и при помощи таких средств как Video for Windows фирмы Microsoft, VideoStudio фирмы Ulead Systems, Inc. или аналогичных создать AVI-файл на основе подготовленных кадров. При этом можно создать клип "одним махом" или добавлять по кадру вручную.

В первом случае необходимо, чтобы файлы имели названия следующего формата:

```
<имя><номер>.BMP
```

причем нумерация файлов должна соответствовать порядковому номеру кадров: frm001.bmp, frm002.bmp, frm003.bmp и т. д. Для преобразования набора файлов-кадров в видео-клип необходимо выполнить команду Image Sequence.

Клавиша вызова (класс CHotKeyCtrl)



Клавиша вызова или элемент управления "горячая клавиша" (hot key control) позволяет пользователю создавать клавиши вызова, т. е. такие комбинации клавиш, нажатие которых приведет к немедленному выполнению заданной команды.[1] Данный элемент управления отображает пользовательский выбор и проверяет его допустимость. Класс библиотеки MFC *CHotKeyCtrl* обеспечивает интерфейс с описываемым элементом управления.

Собственно комбинация клавиш определяется, во-первых, виртуальным кодом одной из алфавитно-цифровых клавиш, клавиш перемещения курсора и любых других, за исключением только трех клавиш — <Alt>, <Shift> и <Ctrl>, которые являются модификаторами и представляют собой вторую составляющую комбинации. Причем в комбинацию могут быть включены только одна из этих клавиш, любые две или все три.

Для использования описываемого элемента управления необходимо либо включить соответствующий элемент в шаблон блока диалога, либо создать с использованием функции *Create* непосредственно дочернее окно в любом другом. (Класс *CHotKeyCtrl* имеет конструктор без параметров, который просто создает объект класса, ничего более не предпринимая, т. е. никак не взаимодействует с операционной системой.)

BOOL CHotKeyCtrl::**Create**(
  DWORD dwStyle,
  const RECT & rect,
  CWnd* pParentWnd,
  UINT nID) —

возвращает TRUE в случае успешного создания и FALSE в противном случае. Параметр dwStyle содержит любую комбинацию стилей элементов управления. Параметр rect определяет местоположение и размер элемента управления. Параметр pParentWnd должен указывать на родительское окно, причем этот параметр не должен быть равен NULL, что разрешено для некоторых окон в ряде случаев[2]. Параметр nID обеспечивает окно идентификатором, использование которого позволяет взаимодействовать с окном.

---

[1] В рамках Win32s этот элемент управления не поддерживается.

[2] Ряд классов окон не требует указания родительского окна, например, это относится к классам блоков диалога, т. к. при создании соответствующая функция *Create* самостоятельно определяет родительское окно, выбирая в случае необходимости главное окно приложения (вызов функции *AfxGetMainWnd*) или главное системное окно (desktop window).

После создания элемента управления его можно и обычно нужно инициализировать некоторым значением. Специфика использования настраиваемых клавиш вызова требует сохранения информации о конкретной комбинации вне исполняемого модуля или библиотеки динамической компоновки. Это может быть INI-файл, запись в реестре или файл конфигурации, формат которого определяет приложение. Однако начальные (исходные) значения могут находиться как в исполняемом модуле (или DLL), так и в специальных конфигурационных файлах. Действительно, ограничений на изменение без ведома приложения и в INI-файлах и в реестре практически нет даже в рамках Windows NT. Так как собственно механизм сохранения какой бы то ни было информации не является предметом данного раздела, в сопровождающем его примере нет кода, обеспечивающего сохранения текущих назначений. Однако во время работы приложения-примера эти значения сохраняются прямо в памяти с использованием глобального массива.

Итак, для начальной инициализации необходимо воспользоваться функцией *SetHotKey*. Вот ее определение:

```
void CHotKeyCtrl::SetHotKey(
  WORD wVirtualKeyCode,
  WORD wModifiers)
```

Прежде чем описывать параметры, представим и обратную функцию получения информации о выбранной комбинации:

```
void CHotKeyCtrl::GetHotKey(
  WORD& wVirtualKeyCode,
  WORD& wModifiers)
```

Первый параметр определяет виртуальный код клавиш (напомним, что такой клавишей может быть любая за исключением трех клавиш-модификаторов <Alt>, <Shift> и <Ctrl>). Второй параметр как раз и определяет режим нажатия клавиш-модификаторов.

Обратите внимание, что оба параметра функции *GetHotKey* являются ссылками, что вполне естественно, т. к. они используются для получения информации.

Библиотека классов MFC определяет следующие флаги, для определения нажатия клавиш-модификаторов:

HOTKEYF_ALT	клавиша <Alt>
HOTKEYF_CONTROL	клавиша <Ctrl>
HOTKEYF_EXT	в качестве клавиши (помимо модификатора) была нажата одна из клавиш перемещения курсора
HOTKEYF_SHIFT	клавиша <Shift>

Класс *CHotKeyCtrl* имеет в своем составе также еще одну версию функции *GetHotKey*:

DWORD CHotKeyCtrl::**GetHotKey**()

которая не принимает параметров, а наоборот, возвращает и виртуальный код клавиши и модификатор, причем виртуальный код находится в младшем байте младшего слова, модификатор — в старшем байте того же слова, а для использования полученного значения можно воспользоваться следующим кодом, где переменная *m_HotKey* является объектом класса *CHotKeyCtrl*, связанным с соответствующим объектом Windows:

```
...
DWORD dwKeyAndShift = m_HotKey.GetHotKey();
...
m_HotKey.SetHotKey((LOBYTE(LOWORD(dwKeyAndShift)),
  HIBYTE(LOWORD(dwKeyAndShift)));
...
```

Естественно, что далеко не все комбинации клавиш разрешены, так при выборе части комбинаций генерируется звуковой сигнал, что, однако, не мешает их назначению и использованию в качестве клавиш вызова. Win32 API и в месте с ним класс *CHotKeyCtrl* дают возможность заранее отсечь часть комбинаций, для чего можно воспользоваться функцией *SetRules*, которая определяет ряд правил выбора комбинаций для последующего назначения их в качестве клавиш вызова:

void CHotKeyCtrl::**SetRules**(
  WORD wInvalidComb,
  WORD wModifiers)

Параметр wInvalidComb собственно определяет такие правила, wModifiers позволяет указать флаги, определяющие клавиши-модификаторы, которые будут использованы вместо запрещенных в случае их выбора.

Параметр *wInvalidComb* может принимать любую комбинацию следующих флагов, запрещая выбор тех или иных клавиш-модификаторов:

HKCOMB_A	запрет выбора клавиши <Alt>
HKCOMB_C	запрет выбора клавиши <Ctrl>
HKCOMB_CA	запрет выбора клавиш <Ctrl> и <Alt> одновременно
HKCOMB_NONE	запрет отсутствия выбора клавиш-модификаторов
HKCOMB_S	запрет выбора клавиши <Shift>
HKCOMB_SA	запрет выбора клавиш <Shift>+<Alt> одновременно
HKCOMB_SC	запрет выбора клавиш <Shift>+<Ctrl> одновременно
HKCOMB_SCA	запрет выбора клавиш <Shift>+<Ctrl>+<Alt> одновременно

К сожалению, ни элемент управления, ни класс *CHotKeyCtrl* не имеют средств к связыванию определенных действий с заданной комбинацией

клавиш. Для этого придется воспользоваться или посылкой специального сообщения WM_SETHOTKEY или функцией Win32 API *RegisterHotKey*. Однако выбор того или иного способа обуславливается отнюдь не удобством — каждый из способов регистрирует в системе определенные типы сообщений для комбинаций клавиш.

Microsoft предлагает следующие наименования для типов клавиш вызова — *глобальные* и *поточно-ориентированные*. Рассмотрим эти типы более подробно.

Глобальные клавиши вызова

Глобальные клавиши вызова ассоциируются только с окнами верхнего уровня и позволяют их активизировать в любой момент. Для регистрации горячей клавиши такого типа необходимо послать сообщение WM_SETHOTKEY нужному окну. Активизация приложения происходит следующим образом — окну посылается сообщение WM_SYSCOMMAND с идентификатором сообщения SC_HOTKEY. Больше никакой дополнительной информации оконная процедура, обслуживающая сообщение WM_SYSCOMMAND, не получает. Таким образом, имеет смысл определить только одну комбинацию клавиш для создания горячей клавиши этого типа. Отметим, что система отслеживает назначенную горячую клавишу до тех пор, пока приложение, пославшее сообщение WM_SETHOTKEY, активно.

Поточно-ориентированные клавиши вызова

В отличие от глобальных, поточно-ориентированные клавиши вызова могут быть назначены для каждого интерактивного потока приложения, т. е. такого потока, который представлен окном. Для назначения горячей клавиши необходимо воспользоваться функцией Win32 API *RegisterHotKey*. При нажатии заданной комбинации клавиш Windows помещает в начало очереди сообщений соответствующего потока сообщение WM_HOTKEY, которое также содержит полную информацию о кодах нажатия, и, что более важно, определенный пользователем идентификатор.

Важность функции *RegisterHotKey* для изложения материала требует приведения ее описания в этом разделе.

```
BOOL RegisterHotKey(
  HWND hWnd,
  int id,
  UINT fsModifiers,
  UINT vk)
```

Параметр hWnd определяет дескриптор окна, которому будет отправлено сообщение WM_HOTKEY, при значении этого параметра NULL сообщение будет направляться в очередь вызвавшего функцию потока. Остальные три параметра определяют идентификатор и коды, определяющие комбинацию клавиш.

На значение идентификатора накладываются определенные ограничения: для исполняемых модулей значения идентификаторов должны лежать в пределах от 0x0000 и до 0xBFFF, а код, находящийся в динамически подключаемых библиотеках (DLL), должен определять значения идентификаторов в пределах от 0xC000 и до 0xFFFF.[1]

В случае, если задаваемая комбинация клавиш имеет тот же идентификатор, что и заданная ранее, последнее значение заменяется новым. Для отмены горячей клавиши необходимо воспользоваться функцией *UnregisterHotKey*.

Функция *RegisterHotKey* принимает в качестве флагов, определяющих клавиши-модификаторы, значения, отличные от тех, что получает приложение в результате выполнения функции *CHotKeyCtrl::GetHotKey*. Поэтому перед регистрацией горячей клавиши необходимо провести переопределение. Далее приведем мнемоники флагов, используемых в функции *RegisterHotKey*.

MOD_ALT	клавиша <Alt>
MOD_CONTROL	клавиша <Ctrl>
MOD_SHIFT	клавиша <Shift>

Необходимо отметить, что и глобальные и поточно-ориентированные клавиши вызова имеют приоритет перед комбинациями, заданными в качестве ускорителей (acceierators) выполнения команд.

Пример использования класса CHotKeyCtrl

В данном примере есть возможность назначать пять клавиш вызова, из которых четыре являются поточно-зависимыми, а одна — глобальной. Все назначения происходят в функциях-членах класса CHotKeyDlg, создающего блок диалога. Последний выводится на экран по команде IDM_HOTKEY, которая добавлена в системное меню, т. к. само приложение меню не имеет. Обратите внимание, что код этой команды должен лежать в определенных пределах.

Для хранения начальных значений комбинаций клавиш и соответствующих команд, а так же для хранения новых устанавливаемых значений комбинаций клавиш используется глобальный массив gHotKey. При каждом создании блока диалога (функция CHotKeyDlg::OnInitDialog) все пять элементов управления инициализируются последними значениями комбинаций клавиш, а при завершении диалога, при подтверждении введенных значений (функция CHotKeyDlg::OnOK) будет обновлено содержимое массива, а также зарегистрированы все пять клавиш вызова.

---

[1] Это ограничение связано с тем, что для исключения конфликтов значение идентификатора следует определять при помощи функции Win32 API *GlobalAddAtom*, которая как раз и возвращает значения в этих пределах.

Как для создания блока диалога в ответ на команду, так и для обработки команды SC_HOTKEY от глобальной горячей клавиши, класс окна CHotKeyFrame содержит обработчик сообщения WM_SYSCOMMAND. В коде, обрабатывающем данную горячую клавишу, предусмотрен вывод текущего времени по центру окна.

Для обработки команд, связанных с поточно-ориентированными клавишами вызова, класс окна имеет функцию CHotKeyFrame::OnHotKey), в которой в соответствии с полученным идентификатором производятся манипуляции с главным окном приложения — оно может быть минимизировано, максимизировано, восстановлено в размерах и спрятано.

```
/*************************************************************
Файл HotKeyFrm.cpp
Copyright (c) 1997 Мешков А., Тихомиров Ю., Шведов Д.
*************************************************************/

// HotKeyFrm.cpp : implementation file
//

#include "stdafx.h"
#include "HotKey.h"
#include "HotKeyFrm.h"

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

// код команды вызова диалога для назначения клавиш вызова
#define IDM_HOTKEYBOX  0x0010

// массив структур для хранения комбинаций клавиш и
// идентификаторов команд, связанных с ними. Первые четыре
// клавиши вызова являются поточно-ориентированными,
// а последняя — глобальной и соответственно не имеет
// связанного с ней идентификатора
static struct {
  WORD m_wVK;
  WORD m_wMod;
  UINT m_nCmd;
} gHotKey[5] =
{{VK_F1, HOTKEYF_SHIFT | HOTKEYF_CONTROL, SW_MINIMIZE},
 {VK_F2, HOTKEYF_SHIFT | HOTKEYF_CONTROL, SW_MAXIMIZE},
 {VK_F3, HOTKEYF_SHIFT | HOTKEYF_CONTROL, SW_SHOWNORMAL},
 {VK_F4, HOTKEYF_SHIFT | HOTKEYF_CONTROL, SW_HIDE},
 {VK_F5, HOTKEYF_SHIFT | HOTKEYF_CONTROL}
};
```

```
//
// функция перевода кодов флагов для использования
// в функции RegisterHotKey
//
UINT TransModifiers(WORD wMod)
{
  UINT uMod = 0;

  if (wMod & HOTKEYF_ALT)
    uMod |= MOD_ALT;
  if (wMod & HOTKEYF_SHIFT)
    uMod |= MOD_SHIFT;
  if (wMod & HOTKEYF_CONTROL)
    uMod |= MOD_CONTROL;

  return uMod;
}

/////////////////////////////////////////////////////////
// CHotKeyDlg - диалог "Клавиши вызова"

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

  // Dialog Data
    //{{AFX_DATA(CHotKeyDlg)
    enum { IDD = IDD_HOTKEYBOX };
    CHotKeyCtrl m_Minimize;
    CHotKeyCtrl m_Maximize;
    CHotKeyCtrl m_ShowNormal;
    CHotKeyCtrl m_Hide;
    CHotKeyCtrl m_DrawTime;
    //}}AFX_DATA

  // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CHotKeyDlg)
    protected:
    // DDX/DDV support
    virtual void DoDataExchange(CDataExchange* pDX);
    //}}AFX_VIRTUAL

// Implementation
protected:
  //{{AFX_MSG(CHotKeyDlg)
  virtual void OnOK();
  virtual BOOL OnInitDialog();
  //}}AFX_MSG
  DECLARE_MESSAGE_MAP()
};
```

```
CHotKeyDlg::CHotKeyDlg() : CDialog(CHotKeyDlg::IDD)
{
  //{{AFX_DATA_INIT(CHotKeyDlg)
  //}}AFX_DATA_INIT
}

void CHotKeyDlg::DoDataExchange(CDataExchange* pDX)
{
  CDialog::DoDataExchange(pDX);
  //{{AFX_DATA_MAP(CHotKeyDlg)
  DDX_Control(pDX, IDC_MINIMIZE, m_Minimize);
  DDX_Control(pDX, IDC_MAXIMIZE, m_Maximize);
  DDX_Control(pDX, IDC_SHOWNORMAL, m_ShowNormal);
  DDX_Control(pDX, IDC_HIDE, m_Hide);
  DDX_Control(pDX, IDC_DRAWTIME, m_DrawTime);
  //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CHotKeyDlg, CDialog)
  //{{AFX_MSG_MAP(CHotKeyDlg)
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()

//
//  функция, в которой проводится начальная инициализация
//  элементов управления
//
BOOL CHotKeyDlg::OnInitDialog()
{
  CDialog::OnInitDialog();

  m_Minimize.SetHotKey(gHotKey[0].m_wVK, gHotKey[0].m_wMod);
  m_Maximize.SetHotKey(gHotKey[1].m_wVK, gHotKey[1].m_wMod);
  m_ShowNormal.SetHotKey(gHotKey[2].m_wVK, gHotKey[2].m_wMod);
  m_Hide.SetHotKey(gHotKey[3].m_wVK, gHotKey[3].m_wMod);
  m_DrawTime.SetHotKey(gHotKey[4].m_wVK, gHotKey[4].m_wMod);

  return TRUE;
  // return TRUE unless you set the focus to a control
  // EXCEPTION: OCX Property Pages should return FALSE
}

//
// в случае подтверждения выбора запоминаем новые
// значения для кодов комбинаций клавиш и
// регистрируем клавиши вызова
//
void CHotKeyDlg::OnOK()
{
  m_Minimize.GetHotKey(gHotKey[0].m_wVK, gHotKey[0].m_wMod);
```

```
  RegisterHotKey(AfxGetMainWnd()->m_hWnd, 0,
    TransModifiers(gHotKey[0].m_wMod), gHotKey[0].m_wVK);

  m_Maximize.GetHotKey(gHotKey[1].m_wVK, gHotKey[1].m_wMod);
  RegisterHotKey(AfxGetMainWnd()->m_hWnd, 1,
    TransModifiers(gHotKey[1].m_wMod), gHotKey[1].m_wVK);

  m_ShowNormal.GetHotKey(gHotKey[2].m_wVK, gHotKey[2].m_wMod);
  RegisterHotKey(AfxGetMainWnd()->m_hWnd, 2,
    TransModifiers(gHotKey[2].m_wMod), gHotKey[2].m_wVK);

  m_Hide.GetHotKey(gHotKey[3].m_wVK, gHotKey[3].m_wMod);
  RegisterHotKey(AfxGetMainWnd()->m_hWnd, 3,
    TransModifiers(gHotKey[3].m_wMod), gHotKey[3].m_wVK);

  m_DrawTime.GetHotKey(gHotKey[4].m_wVK, gHotKey[4].m_wMod);

  DWORD dwKeyAndShift = m_DrawTime.GetHotKey();
  AfxGetMainWnd()->SendMessage(WM_SETHOTKEY, dwKeyAndShift);

  CDialog::OnOK();
}

/////////////////////////////////////////////////////////////
// CHotKeyFrame

CHotKeyFrame::CHotKeyFrame()
{
}

CHotKeyFrame::~CHotKeyFrame()
{
}

BEGIN_MESSAGE_MAP(CHotKeyFrame, CFrameWnd)
  //{{AFX_MSG_MAP(CHotKeyFrame)
  ON_WM_PAINT()
  ON_WM_SYSCOMMAND()
  ON_WM_CREATE()
  ON_MESSAGE(WM_HOTKEY, OnHotKey)
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////
// CHotKeyFrame message handlers

void CHotKeyFrame::OnPaint()
{
  CPaintDC dc(this);
}

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

```
  //
  // добавляем пункт в системное меню
  //

  CMenu* pSysMenu = GetSystemMenu(FALSE);

  // IDM_HOTKEYBOX должен находиться в заданных пределах
  ASSERT((IDM_HOTKEYBOX & 0xFFF0) == IDM_HOTKEYBOX);
  ASSERT(IDM_HOTKEYBOX < 0xF000);

  pSysMenu->AppendMenu(MF_STRING, IDM_HOTKEYBOX,
    _T("Клавиши вызова"));

  return 0;
}

//
// функция обработки сообщения WM_SYSCOMMAND
//
void CHotKeyFrame::OnSysCommand(UINT nID, LPARAM lParam)
{
  if ((nID & 0xFFF0) == IDM_HOTKEYBOX)
  {
    CHotKeyDlg dlgHotKey;
    dlgHotKey.DoModal();
  }
  else if (nID == SC_HOTKEY)
  {
    if (!IsIconic())
    {
      InvalidateRect(NULL);
      UpdateWindow();

      CClientDC dc(this);

      CRect rect;
      GetClientRect(rect);

      dc.SetBkMode(TRANSPARENT);
      dc.DrawText(CTime::GetCurrentTime().Format(_T("%H:%M:%S")),
        &rect, DT_CENTER | DT_VCENTER | DT_SINGLELINE);
    }
  }
  else
    CWnd::OnSysCommand(nID, lParam);
}

//
// функция обработки сообщения WM_HOTKEY
//
LONG CHotKeyFrame::OnHotKey(UINT nID, LPARAM lParam)
```

```
{
  if (0 <= nID && nID < 5)
    ShowWindow(gHotKey[nID].m_nCmd);

  return 0;
}
```

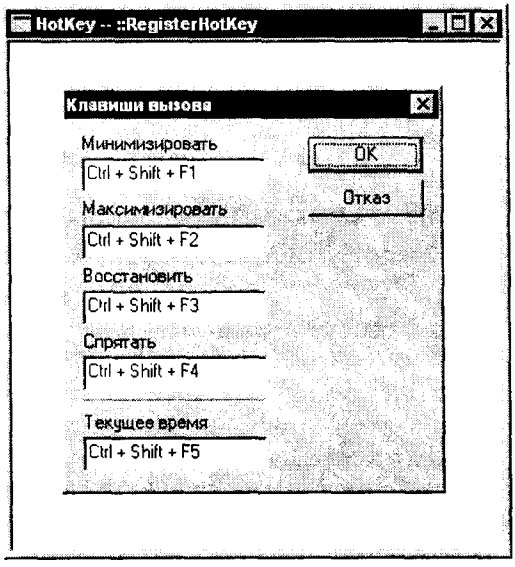
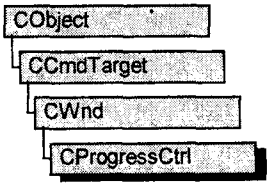


Рис. 24. Блок диалога настройки клавиш вызова в главном окне приложения HotKey

Индикатор (класс CProgressCtrl)



Индикатор — это дочернее окно, которое можно использовать в приложении для индикации процесса выполнения некоторой длительной операции. Такое окно представляет собой вытянутую и расположенную горизонтально прямоугольную область (полосу). По мере выполнения операции содержимое области заполняется слева направо прямоугольниками, цвет которых соответствует системному цвету, используемому для подсвечивания надписей или иных объектов.

В отличие от большинства других элементов управления, цвет которых можно задать в ответ на соответствующий запрос, цвет индикатора определяется системными установками и изменению не подлежит.

Большинство программ индикацию реализуют двумя способами: добавляют элемент индикации в блок диалога, либо используют его в специально созданных панелях диалога, которые были описаны в главе 13 первого тома

книги. И тот и другой способ подразумевает использование шаблонов блока диалога, что исключает необходимость использования функции создания окна индикатора. Библиотека классов MFC имеет в своем составе класс *CProgressCtrl*, который обеспечивает интерфейс между приложением и объектом Windows "индикатор". Этот класс имеет конструктор, задача которого — создание объекта класса, собственно же элемент индикации может быть создан при помощи функции *Create* (использование которой не актуально при задании элемента в шаблоне блока диалога):

```
BOOL CProgressCtrl::Create(
  DWORD dwStyle,
  const RECT & rect,
  CWnd* pParentWnd,
  UINT nID)
```

Параметры этой функции полностью совпадают с рассмотренными ранее для функции CHotKeyCtrl::Create.

Реализация процесса, а также его взаимосвязь с индикацией полностью лежат на разработчике приложения. Собственно индикатор является лишь средством отображения отношения двух чисел. Первое определяется величиной некоторого интервала, а второе — смещением от начала интервала. Границы интервала лежат в пределах от 0 и до 65535. А смещение определяется заданным текущим абсолютным значением и нижней границей интервала. Возможность задания интервала и смещения его нижней границы относительно нуля может обеспечить некоторую связь с природой отслеживаемого процесса, исключая дополнительный код на преобразование.

Для задания интервала класс *CProgressCtrl* имеет функцию *SetRange*:

```
void CProgressCtrl::SetRange(
  int nLower,
  int nUpper)
```

где параметры *nLower* и *nUpper* определяют его нижнюю и верхнюю границы соответственно.

Текущее значение индикатора может быть установлено тремя различными способами. Во-первых, можно непосредственно задать текущее значение:

```
int CProgressCtrl::SetPos(int nPos)
```

Эта функция не только задает текущее значение (параметр nPos), но также в качестве результата выполнения возвращает предыдущее значение индикатора. При неверном задании текущего значения (вне интервала) произойдет приведение нового значения к значению ближайшей границы интервала.

Во-вторых, осуществить ранее заданное смещение (шаг) относительно текущего значения при помощи функции *StepIt*:

```
int CProgressCtrl::StepIt()
```

Эта функция также возвращает предыдущее значение индикатора. Причем в случае, если в результате добавления к текущему значению величины шага значение индикатора выйдет за пределы интервала, новое значение индикатора

будет возвращено в заданные пределы. Например, на интервале от 10 до 100, при текущем значении индикатора 90 и величине шага 20 после выполнения функции StepIt текущее значение индикатора будет равным 20, на том же интервале при текущем значении 20 и величине шага 30 в результате выполнения функции StepIt текущее значение индикатора будет равным 80.

Шаг задается функцией *SetStep*:

int CProgressCtrl::**SetStep**(int nStep)

Задание шага осуществляется при помощи параметра nStep, сама же функция в качестве результата выполнения возвращает предыдущее значение шага. Значение шага может быть отрицательным, что позволяет не только увеличивать значение индикатора, но и уменьшать его.

В-третьих, текущее значение устанавливается непосредственно заданным смещением при помощи функции *OffsetPos*:

int CProgressCtrl::**OffsetPos**(int nPos)

где параметр nPos определяет смещение относительно текущего значения. Сама функция, как и все остальные изменяющие текущее значение, возвращает его предыдущее значение. При выходе за пределы интервала, в отличие от функции StepIt, выполнение функции OffsetPos приведет к тому, что текущим значением интервала будет либо нижняя, либо верхняя границы в зависимости от значения смещения.

При выполнении любой функции, в результате которой может измениться отношение величины интервала и текущего значения (*SetRange*, *SetPos*, *StepIt*, *OffsetPos*) происходит автоматическая перерисовка индикатора.

Однако уже при создании (с использованием функции *Create* или во время создания блока диалога) индикатор инициализируется следующими начальными значениями (по умолчанию):

- текущее значение 0
- интервал в пределах от 0 до 100
- шаг приращения 10

Использование индикатора в панели диалога

Суть следующего примера в том, что индикатор размещен в панели диалога, созданной при помощи класса *CDialogBar*, о котором было рассказано в главе 13 первого тома. При создании объекта CInProgFrame кроме панели диалога также создается строка состояния, которая заменяется время от времени панелью с индикатором. Для этого создаются два таймера. Задача первого — имитация начала и окончания некоторой операции, задача второго — имитация собственно операции. Обработчик сообщений от таймера либо переключает панели, либо изменяет значение индикатора. Для наглядности в код "начала операции" добавлены функции начальной инициализации индикатора.

```
/*************************************************************
Файл InProgFrm.cpp
Copyright (c) 1997 Мешков А., Тихомиров Ю., Шведов Д.
*************************************************************/

// InProgFrm.cpp : implementation file
//
...
// идентификаторы таймеров
#define ID_PULSE   1
#define ID_INPROG  2

// массив настроек строки состояния
static UINT BASED_CODE indicators[] =
{
  ID_SEPARATOR,
  ID_INDICATOR_NUM,
  ID_INDICATOR_CAPS,
  ID_INDICATOR_OVR
};

////////////////////////////////////////////////////////////////
// CInProgFrame
...
// перегружаем функцию PreCreateWindow для задания
// своего класса окна,
// задавая иной фон и отказываясь от стилей CS_VREDRAW/CS_HREDRAW,
// чтобы исключить перерисовку панелей элементов управления
// при изменении размеров главного окна
BOOL CInProgFrame::PreCreateWindow(CREATESTRUCT& cs)
{
  cs.lpszClass = AfxRegisterWndClass(CS_DBLCLKS,
    AfxGetApp()->LoadStandardCursor(IDC_ARROW),
    (HBRUSH)(COLOR_WINDOW+1),
    AfxGetApp()->LoadIcon(IDR_INPROG));

  return TRUE;
}
...
int CInProgFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
  if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
    return -1;

  // создание строки состояния
  if (!m_wndStatusBar.Create(this) ||
      !m_wndStatusBar.SetIndicators(indicators,
      sizeof(indicators)/sizeof(UINT)))
```

```
  {
    TRACE0("Ошибка при создании панели \n");
    return FALSE;
  }

  // скрываем строку состояния - на ее месте будет
  // панель диалога
  ShowControlBar(&m_wndStatusBar, FALSE, FALSE);

  // создаем панель диалога
  if (!m_wndDlgBar.Create(this, IDD_PROGRESS,
    CBRS_BOTTOM, IDD_PROGRESS))
  {
    TRACE0("Ошибка при создании панели \n");
    return -1;
  }

  // разрешаем привязку панелей элементов управления
  EnableDocking(CBRS_ALIGN_BOTTOM);

  // получаем указатель на объект класса CProgressCtrl
  // по его идентификатору в шаблоне блока диалога
  CProgressCtrl *pProgress =
    (CProgressCtrl *)m_wndDlgBar.GetDlgItem(IDC_PROGRESS);

  // инициализируем индикатор
  pProgress->SetRange(0, 1000);
  pProgress->SetPos(0);
  pProgress->SetStep(10);

  // создаем два таймера для имитации длительной операции
  SetTimer(ID_PULSE, 10000, NULL);
  SetTimer(ID_INPROG,  100, NULL);

  return 0;
}

//
// функция-обработчик сообщений от таймера
//
void CInProgFrame::OnTimer(UINT nIDEvent)
{
  // статическая переменная-переключатель или
  // флаг режима
  static int nCheck = 1;

  CProgressCtrl *pProgress =
    (CProgressCtrl *)m_wndDlgBar.GetDlgItem(IDC_PROGRESS);

  if (nIDEvent == ID_PULSE)
  {
    if (nCheck = !nCheck)
```

```
    {
      //
      // начало "операции"
      //

      pProgress->SetRange(0, 1000);
      pProgress->SetPos(0);
      pProgress->SetStep(10);

      ShowControlBar(&m_wndStatusBar, FALSE, FALSE);
      ShowControlBar(&m_wndDlgBar, TRUE, FALSE);
    }
    else
    {
      //
      // завершение "операции"
      //

      ShowControlBar(&m_wndDlgBar, FALSE, FALSE);
      ShowControlBar(&m_wndStatusBar, TRUE, FALSE);
    }
  }
  else if (nIDEvent == ID_INPROG && nCheck)
  {
    //
    // "операция" осуществляется
    //

    pProgress->StepIt();
  }

  CFrameWnd::OnTimer(nIDEvent);
}
```

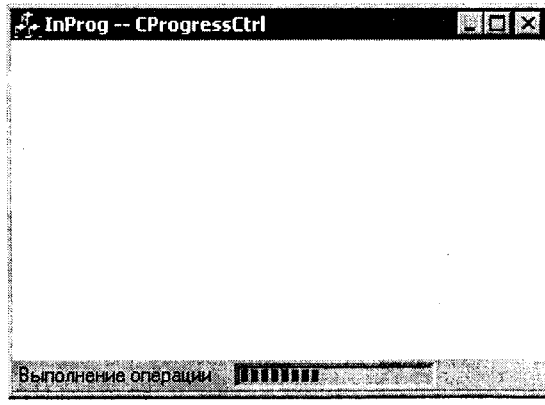
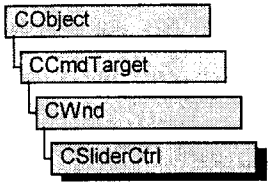
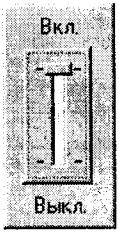


Рис. 25. Окно приложения InProg с индикатором в диалоговой панели

Линейка с бегунком (класс CSliderCtrl)



Линейка с бегунком (SLIDER или TRACKBAR) представляет собой окно с линейкой, бегунком на ней и, возможно, метками, кроме того, можно выделить любой диапазон в рамках размера линейки, которая может иметь как горизонтальную, так и вертикальную ориентацию. Когда пользователь перемещает бегунок при помощи мыши или используя клавиши управления курсором, этот элемент управления посылает уведомляющие сообщения, позволяющие отслеживать производимые изменения.



Этот элемент управления предназначен для того, чтобы пользователи могли выбирать дискретные значения в заданном диапазоне. Вид элемента позволяет использовать его, например, для запуска какой-либо операции, в качестве, если хотите, рубильника.

Библиотека MFC предоставляет класс *CSliderCtrl*, который инкапсулирует возможности линейки с бегунком, упрощая управление этим элементом.

Несмотря на явно аналоговое представление значений, бегунок может принимать значения, лежащие в заданном целочисленном диапазоне. После его создания в рамках блока диалога или в другом окне, можно задать ряд параметров, для чего класс предоставляет большое количество функций. Но об этом позже. Сначала опишем, какие уведомляющие сообщения и в каких случаях посылает данный элемент управления.

Уведомления

Независимо от того, как ориентирован элемент управления — горизонтально или вертикально, — уведомления он посылает родительскому окну в рамках сообщения WM_HSCROLL. Для обработки сообщений от линейки с бегунком следует перегрузить функцию класса родительского окна *OnHScroll*. Эта функция получает код уведомления и указатель на объект *CSliderCtrl*. Следует иметь в виду, что соответствующий параметр функции имеет тип указателя на класс *CScrollCtrl*, поэтому для вызова функций-членов класса *CSliderCtrl* следует произвести преобразование типа.

Все уведомляющие сообщения можно разделить на следующие группы. Сообщения TB_BOTTOM, TB_LINEDOWN, TB_LINEUP и TB_TOP посылаются только при взаимодействии пользователя с элементом управления с помощью клавиатуры, т. е. клавиш перемещения курсора. Сообщения TB_THUMBPOSITION и TB_THUMBTRACK посылаются при действиях мышью. А сообщения TB_ENDTRACK, TB_PAGEDOWN и TB_PAGEUP посылаются в обоих случаях.

Далее в таблице перечислены все уведомления и соответствующие им виртуальные клавиши, вызывающие эти сообщения:

TB_BOTTOM	VK_END — бегунок переводится в самое правое или самое нижнее положение
TB_ENDTRACK	WM_KEYUP — завершение перемещения бегунка
TB_LINEDOWN	VK_RIGHT или VK_DOWN — перемещение бегунка на одну позицию вправо (вниз)[1]
TB_LINEUP	VK_LEFT или VK_UP — перемещение на одну позицию влево (вверх)
TB_PAGEDOWN	VK_NEXT — перемещение на одну "страницу"[2] вправо (вниз) при использовании клавиши <Page Down> или мыши
TB_PAGEUP	VK_PRIOR — перемещение на одну "страницу" влево (вверх) при использовании клавиши <Page Up> или мыши
TB_THUMBPOSITION	WM_LBUTTONUP — текущее положение бегунка после завершения его перемещения мышью; следует за сообщением TB_THUMBTRACK
TB_THUMBTRACK	бегунок перемещался мышью
TB_TOP	VK_HOME — бегунок переводится в самое левое (самое верхнее) положение

Создание элемента управления

Конструктор класса создает только объект *CSliderCtrl*, для создания элемента управления следует вызвать функцию *Create*:

```
BOOL CSliderCtrl::Create(
   DWORD dwStyle,
   const RECT& rect,
   CWnd* pParentWnd,
   UINT nID)
```

Параметр *dwStyle* может содержать любые комбинации стилей элементов управления, включая следующие, свойственные для рассматриваемого элемента управления:

TBS_HORZ	линейка имеет горизонтальную ориентацию (принята по умолчанию)
TBS_VERT	линейка имеет вертикальную ориентацию

---

[1] По умолчанию линейка имеет 101 позицию (от 0 и до 100).

[2] По умолчанию линейка разбивается на 5 "страниц" шириной в 20 позиции.

TBS_AUTOTICKS	линейка имеет метки для всех значений в заданном диапазоне значений; без этого стиля линейка может иметь метки только для начального и конечного положения бегунка или там, где они заданы функциями *SetTic* и *SetTicFreq*; метки добавляются автоматически, поэтому при наличии данного стиля нельзя использовать функции *SetTic* и *SetTicFreq*, для удаления меток можно воспользоваться функцией *ClearTics*; метки устанавливаются сразу же после вызова функции *SetRange*
TBS_NOTICKS	исключает отображение каких-либо меток, в том числе и для начального и конечного положений бегунка
TBS_BOTTOM	линейка имеет метку для конечного положения бегунка
TBS_TOP	линейка имеет метку для начального положения бегунка
TBS_RIGHT	элемент отображает метки справа (снизу) от линейки, бегунок направлен в ту же сторону
TBS_LEFT	элемент отображает метки слева (сверху) от линейки, бегунок направлен в ту же сторону
TBS_BOTH	элемент отображает метки с обеих сторон, бегунок имеет прямоугольную форму, т. е. он фактически не направлен ни в одну из сторон
TBS_ENABLESELRANGE	элемент отображает "допустимый" диапазон выбора; при этом линейка имеет внутреннюю область большего, чем обычно, размера, в которой заданный диапазон имеет цвет, отличный от цвета линейки; начало и конец диапазона ограничены метками треугольной формы

Параметр *rect* задает размер и расположение линейки. Может быть объектом *CRect* или структурой *RECT*, в случае, если заданный размер меньше, чем требуется для отображения линейки и бегунка, масштабирование их не производится — часть элемента управления будет скрыта. Параметр *pParentWnd* задает родительское окно (обычно это *CDialog*). Значение этого параметра не должно быть равно NULL. Параметр *nID* задает идентификатор линейки.

Параметры элемента управления

Кроме возможности задания стилей, можно получать текущие значения установок, а также изменять (настраивать) следующие значения (напомним, что по умолчанию линейка имеет 101 позицию и пять "страниц")[1]:

---

[1] Некоторые из функций настройки по умолчанию после изменения параметров не допускают перерисовки линейки в соответствии с новыми значениями. Это может привести к неожиданным результатам, т. к. даже изменение положения бегунка приводит лишь к перерисовке области, непосредственно располагающейся под бегунком. Таким образом, отказ от перерисовки при смене сразу нескольких параметров следует завершать сообщением о необходимости обновления элемента управления (например, вызовом функции *InvalidateRect*).

- общий диапазон (максимальное и минимальное положение бегунка). Для этого можно воспользоваться функциями *GetRangeMax*, *SetRangeMax*, *GetRangeMin*, *SetRangeMin*, *GetRange* и *SetRange*;
- величину единичных перемещений, уведомление о которых производится сообщениями TB_LINEUP и TB_LINEDOWN (функции *GetLineSize* и *SetLineSize*);
- величину страничных перемещений, уведомления о которых производятся сообщениями TB_PAGEUP и TB_PAGEDOWN (функции *GetPageSize* и *SetPageSize*);
- "допустимый" диапазон (максимальное и минимальное его значения). Для этого можно воспользоваться функциями *GetSelection*, *SetSelection*. Для того чтобы отказаться от отображения "допустимого" диапазона, следует вызвать функцию *ClearSel*;
- размер элемента управления (функция *GetChannelRect*) и размер бегунка (функция *GetThumbRect*).

Отдельно рассмотрим функции, управляющие отображением меток и положением бегунка.

Положение меток

Для получения количества меток следует воспользоваться функцией *GetNumTics*.

Положение метки лежит в общем диапазоне значений. Если стиль элемента управления TBS_AUTOTICKS, количество меток определяется общим диапазоном, несмотря на то, что будущее количество видимых меток, определяется функцией *SetTicFreq*.

Вообще задание этого стиля приводит к тому, что для определения положения следует не пользоваться следующими функциями, а вычислять его.

Для получения положения всех меток класс имеет функцию *GetTicArray*:

DWORD* CSliderCtrl::**GetTicArray**()

которая возвращает указатель на массив, содержащий положения всех меток элемента управления.

Для получения положения конкретной метки по ее номеру, считая слева или сверху, служит функция *GetTic*:

int CSliderCtrl::**GetTic**(int nTic)

которая возвращает значение, равное -1, в случае, если параметр nTic содержит несуществующий номер. А установить положение метки позволяет функция SetTic.

Для получения позиции метки в единицах измерения клиентской области родительского окна следует пользоваться функцией *GetTicPos*.

При задании стиля TBS_AUTOTICKS частота отображения меток определяется функцией *SetTicFreq*:

void CSliderCtrl::**SetTicFreq**(int nFreq)

параметр nFreq сообщает, как часто следует отображать метку. Значение 10, например, говорит о том, что отображению подлежит каждая десятая метка, считая с нулевой. (Крайняя нижняя или правая метка также будет отображена).

Для удаления меток следует воспользоваться функцией *ClearTics*.

Положение бегунка

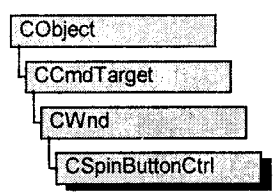
Для получения текущего положения бегунка, а также программной установки его положения следует использовать функции *GetPos* и *SetPos* соответственно.

Класс *CSliderCtrl* имеет в своем составе очень полезную функцию, позволяющую проверять допустимость положения бегунка, ранее установленного при помощи функции *SetPos*:

void CSliderCtrl::**VerifyPos**() —

не возвращает какое-либо значение, позволяющее определить допустимость заданного положения (причем, заданного отнюдь не параметром этой функции!). Работа функции заключается в том, что в случае, когда текущее положение лежит вне общего диапазона, она устанавливает бегунок в крайнее левое (верхнее) или правое (нижнее) положение, в зависимости от того, к какому положению ближе заданное. Этой функцией ни в коем случае *нельзя* пользоваться в обработчике уведомляющих сообщений!

Спин (класс CSpinButtonCtrl)



Элемент управления "спин" (spin button или up-down) представляет собой пару кнопок со стрелками, назначение которых — в смене значения, например, уменьшение либо увеличение числового значения. Спин похож на вырожденную полосу прокрутки (SCROLL BAR) и действует примерно так же. Кроме того, этот элемент управления можно ассоциировать с другим элементом управления так, что его содержимое будет меняться под управлением спина. Такие ассоциированные элементы управления называются *окнами сопровождения* (BUDDY WINDOW).

Если спин ассоциирован с другим окном (элементом управления), для пользователя они выглядят как единый элемент управления. Действительно, спин может автоматически позиционировать себя с другим элементом управления, располагающимся прямо перед ним, например, в шаблоне блока диалога, и самостоятельно устанавливать заголовок этого элемента.

В случае, когда спин меняет числовые значения, нажатие верхней кнопки приводит к смене значения в сторону максимума, а нижней — в сторону минимума. Максимумом по умолчанию является значение 0, а минимумом — значение 100.

Не имея ассоциированного с ним окна, спин функционирует как упрощенный вариант полосы прокрутки. Например, элемент управления, реализующий вкладки (TAB), использует спин для осуществления доступа к дополнительным вкладкам.

Уведомления

Спин может посылать до четырех различных сообщений для одной ориентации. Первые из которых — уведомляющие. Все они посылаются родительскому окну и никак не воздействуют на ассоциированное со спином окно. Эти три сообщения взаимосвязаны.

Первое сообщение, UDN_DELTAPOS, посылается родительскому окну при нажатии одной из кнопок со стрелками для того, чтобы дать возможность приложению до осуществления модификации содержимого ассоциированного со спином окна предпринять какие-либо действия. Например, проверить допустимость изменений, отменить или разрешить их. Для того чтобы отменить изменения в ассоциированном окне, следует в обработчике этого сообщения установить по указателю *pResult* значение TRUE. И наоборот — значение FALSE разрешит произвести изменения:

```
BEGIN_MESSAGE_MAP(CSunDlg, CDialog)
  //{{AFX_MSG_MAP(CSunDlg)
  ...
  ON_NOTIFY(UDN_DELTAPOS, IDC_SPIN, OnDeltaPos)
  ...
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()

void CSunDlg::OnDeltaPos(NMHDR* pNMHDR, LRESULT* pResult)
{
  NM_UPDOWN* pNMUpDown = (NM_UPDOWN*)pNMHDR;

  // меняем значение в два раза быстрее,
  // чем задано
  pNMUpDown->iDelta *= 2;

  // в случае, если ожидаемое значение
  // будет больше 10 — запрещаем изменение
  if (pNMUpDown->iPos + pNMUpDown->iDelta > 10)
    *pResult = TRUE; // запрет изменения
  else
    *pResult = FALSE; // разрешение изменения
}
```

Как видно из примера, обработчик получает указатель на структуру *NM_UPDOWN*, которая содержит необходимую информацию о возможных изменениях:

```
typedef struct _NM_UPDOWN {
  NMHDR hdr;
  int iPos;
  int iDelta;
} NM_UPDOWNW
```

Первый параметр структуры содержит заголовок. Он требуется для всех уведомляющих сообщений. Заголовок содержит дескриптор и идентификатор окна, пославшего сообщение, а также код самого сообщения. Параметр *iPos* содержит текущее значение спина, а параметр *iDelta* — возможное изменение текущего положения. Оба параметра числовые и могут быть отрицательными.

Возникает вопрос: что делать, если ассоциированное окно содержит нечисловые значения, или ассоциированного окна нет, ведь изменение значения спина должно быть связано с выполнением некоторых действий?

Для ответа на него, посмотрим, что же происходит после обработки сообщения UDN_DELTAPOS. В случае, если разрешение изменения получено (а по умолчанию именно так и происходит), спин посылает одно из сообщений, аналогичных сообщениям от полосы прокрутки: WM_HSCROLL или WM_VSCROLL. Тип сообщения зависит от того, как ориентирован спин, вертикально или горизонтально. В обоих случаях обработчики получают всю необходимую информацию.

```
void CSunDlg::OnVScroll(
  UINT nSBCode, UINT nPos, CScrollBar* pScrollBar)
{
  // обрабатываем только сообщения от спина
  if (pScrollBar->m_hWnd == GetDlgItem(IDC_SPIN)->m_hWnd)
  {
    // нас интересует завершение "прокрутки" спина
    if (nSBCode == SB_ENDSCROLL)
    {
      // здесь выполняются необходимые действия,
      // связанные с изменением значения спина
      // текущее значение спина содержит
      // параметр nPos
      ...
    }
  }

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

Автоматическое изменение

При наличии ассоциированного окна спин может самостоятельно менять его содержимое только в одном случае. Когда спин создан для изменения целочисленных значений, при нажатии одной из кнопок со стрелкой, спин посылает ассоциированному с ним окну сообщения WM_GETTEXT и WM_SETTEXT.

Первое сообщение используется для того, чтобы определить текущее значение содержимого окна, а второе — для его изменения. В качестве значения нового содержимого окна выступает строка, содержащая символьное представление числа — текущего значения спина. Такой подход позволяет синхронизировать содержимое окна и значения спина, т. е. позволяет менять содержимое ассоциированного окна и другими средствами помимо спина — программно или вручную пользователем.

А что будет, если пользователь ввел строку текста, которая вовсе не является представлением числа?

В этом случае в качестве текущего значения, которое будет увеличено или уменьшено, выступает *текущее значение* спина — окно ведь не содержит числа. Реализация, пожалуй, довольно логична.

Из вышесказанного следует, что связывать спин можно только с такими типами окон, которые меняют свое содержимое при получении сообщения WM_SETTEXT. Однако это не совсем так — любому окну Windows можно посылать любые сообщения, следовательно, и любое окно может быть связано со спином. Другое дело, что содержимое его может не меняться. Например, кнопки, поля редактирования, статические элементы откликаются на воздействие со стороны спина, а другие, такие как индикатор (PROGRESS BAR), никак не реагируют.

Создание спина

Конструктор класса создает только объект *CSpinButtonCtrl*, для создания элемента управления следует вызвать функцию *Create*:

```
BOOL CSpinButtonCtrl::Create(
   DWORD dwStyle,
   const RECT & rect,
   CWnd* pParentWnd,
   UINT nID)
```

Параметр *dwStyle* может принимать любые комбинации стилей элементов управления плюс следующие, свойственные для рассматриваемого элемента управления:

UDS_HORZ	горизонтальная ориентация спина (по умолчанию — вертикальная)

UDS_WRAP	поддерживается переход через граничные значения; в этом случае после максимального значения текущим становится минимальное и наоборот
UDS_ARROWKEYS	поддерживается управление спином с помощью клавиш перемещения курсора (<↑> и <↓>, независимо от ориентации элемента управления)
UDS_SETBUDDYINT	в качестве значений содержимого ассоциированного окна используются целочисленные значения — изменение содержимого ассоциированного окна происходит автоматически средствами спина (используются сообщения WM_GETTEXT и WM_SETTEXT)
UDS_NOTHOUSANDS	содержимое ассоциированного окна не имеет разделителя тысяч между каждыми тремя цифрами
UDS_AUTOBUDDY	автоматический выбор в качестве ассоциированного окна ближайшего предыдущего в списке созданных, например, в шаблоне блока диалога
UDS_ALIGNRIGHT	спин располагается в правой части ассоциированного окна, уменьшая тем самым его клиентскую область
UDS_ALIGNLEFT	спин располагается в левой части ассоциированного окна, уменьшая тем самым его клиентскую область

Параметр *rect* задает размер и расположение спина. Может быть объектом *CRect* или структурой *RECT*. Параметр *pParentWnd* задает родительское окно (обычно это *CDialog*). Значение этого параметра не должно быть равно NULL. Параметр *nID* задает идентификатор редактора.

Параметры элемента управления

В случае, если спин не имеет стиля UDS_AUTOBUDDY, класс предоставляет возможность связывания с ним окна (функция *SetBuddy* возвращает указатель на объект класса *CWnd* — окно, которое было ранее ассоциировано со спином). В то же время можно получить указатель на объект класса *CWnd* — окно, которое в настоящий момент ассоциировано со спином (функция *GetBuddy*). Указатель может указывать на временный объект, и следовательно, не может быть сохранен для дальнейшего использования.

Для установки значения спина и получения текущего значения используются функции *SetPos* и *GetPos* соответственно. Тип текущего значения — целое число, следовательно для изменения содержимого ассоциированного окна, которое не является числом, следует использовать некоторое преобразование. В этом случае значение спина может использоваться в качестве индекса массива строк или объектов другого типа.

При получении текущего значения спин запрашивает содержимое ассоциированного с ним окна.

Обе функции проверяют заданное значение или результат преобразования содержимого ассоциированного окна. Это значение должно лежать в преде-

лах заданного диапазона. Напомним, что по умолчанию для спина задан диапазон от 100 до 0 (именно так: 100 — минимальное значение, а 0 — максимальное). Для изменения диапазона следует использовать функцию *SetRange*:

```
void CSpinButtonCtrl::SetRange(
  int nLower,
  int nUpper) —
```

параметры nLower и nUpper содержат значения границ диапазона. Значения и того и другого параметра должны лежать в пределах от UD_MINVAL и до UD_MAXVAL. Кроме того, разница между максимальным и минимальным значениями границ не должна быть больше UD_MAXVAL. В настоящий момент эти мнемоники представляют следующие значения:

UD_MAXVAL	0x7FFF (32767)
UD_MINVAL	-UD_MAXVAL (-32767)

> **Примечание**
>
> Естественно, что в программе для проверки допустимости значений следует использовать не числа, а мнемонические обозначения границ.

Кроме назначения новых границ диапазона, можно, воспользовавшись функцией *GetRange*, получить текущие значения границ диапазона изменения спина:

```
DWORD CSpinButtonCtrl::GetRange()
```

и

```
void CSpinButtonCtrl::GetRange(
  int & lower,
  int & upper)
```

Первая версия функции возвращает 32-разрядное число, младшее слово которого содержит верхнюю границу, а старшее — нижнюю границу диапазона допустимых значений.

При необходимости существенного изменения значения спина, а вместе с тем и содержимого ассоциированного окна именно с помощью средств спина, т. е. его кнопок со стрелками, приходится тратить довольно много времени. Приращение значения возможно только на одну единицу. Однако спин предоставляет возможность менять значения приращения в зависимости от времени непрерывного нажатия кнопок. Для этого следует использовать функции *SetAccel* и *GetAccel*:

```
BOOL CSpinButtonCtrl::SetAccel(
  int nAccel,
  UDACCEL* pAccel) —
```

эта функция позволяет задать массив, включающий в качестве элементов структуры *UDACCEL*, которые содержат необходимую информацию для динамического изменения значения приращения:

```
typedef struct {
  int nSec;
  int nInc;
} UDACCEL;
```

поле nSec содержит время в секундах, по истечении которого приращение устанавливается равным значению поля nInc.

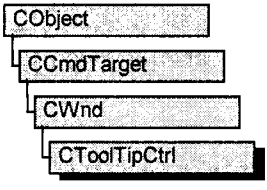
Массив должен быть упорядочен по возрастанию поля nSec, которое содержит время от начала взаимодействия пользователя со спином.

И, наконец, последняя группа функций, позволяющая задавать основание системы счисления для отображения числовых значений спина в ассоциированном окне. Всего можно использовать две системы счисления — десятичную и шестнадцатеричную. Для этого используется, как обычно, пара функций *SetBase* и *GetBase*. Первая позволяет назначать систему счисления, а вторая — получать информацию о текущей установленной:

int CSpinButtonCtrl::**SetBase**(int nBase) —

функция возвращает основание ранее установленной системы счисления, либо 0, если была задана система, не поддерживаемая спином, т. е. не десятичная и не шестнадцатеричная. Параметр nBase может принимать значения 10 или 16. Следует добавить, что при установке шестнадцатеричной системы счисления значения спина *не могут быть отрицательными*.

Всплывающая подсказка (класс CToolTipCtrl)



Элемент управления "всплывающая подсказка" представляет собой небольшое всплывающее окно, отображающее однострочный текст в качестве краткого описания или комментария к связанному с ней объекту приложения. Таким объектом может быть любое окно или даже прямоугольная область клиентской части окна. Большую часть времени подсказка скрыта и появляется лишь тогда, когда пользователь устанавливает курсор мыши на объект, имеющий подсказку, и оставляет его в границах объекта примерно полсекунды. Естественно, что подсказка появляется в непосредственной близости от объекта. Подсказка исчезает при нажатии кнопки мыши или при перемещении курсора вне объекта.

В качестве текста подсказки может использоваться любая необходимая информация, т. к. перед ее выводом элемент управления получает извещение и может сгенерировать текст "на ходу". Например, в случае, если в области уведомления полосы задач задано отображение текущего времени, помещение на часы курсора приведет (через полсекунды) появление подсказки с текущей датой.

Следует добавить, что один элемент управления может обеспечить подсказкой несколько объектов.

Класс *CToolTipCtrl* и соответствующие функции-члены ряда оконных классов библиотеки MFC обеспечивают полную поддержку всплывающих подсказок. Однако сами эти элементы управления реализованы лишь в рамках Windows 95 и Windows NT, начиная с версии 3.51.

Интересным представляется то, как можно реализовать поддержку всплывающих подсказок в приложении. При реализации одного из способов класс *CToolTipCtrl* в явном виде приложению, в принципе, не нужен — класс окна имеет средства, обеспечивающие все необходимое, достаточно перегрузить одну или несколько виртуальных функций. Другой способ, напротив, в полной мере может использовать только объект (объекты) класса *CToolTipCtrl* и при этом никак не задействовать специальные возможности класса *CWnd* и производных от него. Наконец, третий способ — ничего не делать (почти ничего). Правда, при этом подсказками будут снабжены только элементы панели (панелей) инструментов (TOOLBAR) — и все. Кстати сказать, такие средства автоматической генерации кода, как Component Gallery и AppWizard обеспечивают 90% всего необходимого для реализации второго и третьего способов соответственно.

Оконная поддержка

Для использования всплывающих подсказок необходимо проделать целый ряд вспомогательных действий, поэтому прежде чем описывать непосредственно класс *CToolTipCtrl*, рассмотрим механизм, позволяющий обойтись без него. Он довольно сложен, однако часть классов библиотеки MFC поддерживают подсказки автоматически — класс *CFrameWnd* и производные от него (тот самый третий способ). Связано это с тем, что данный класс имеет в своем составе обработчик уведомляющего сообщения TTN_NEEDTEXT. Однако все остальные классы окон, не являющиеся производными от класса *CFrameWnd*, такой поддержки не имеют. Класс окна *CWnd* обеспечивает разработчиков следующими функциями поддержки подсказок: *EnableToolTips*, *CancelToolTips*, *FilterToolTipMessage* и *OnToolHitTest*. Рассмотрим часть из них более подробно.

BOOL CWnd::**EnableToolTips**(BOOL bEnable) —
возвращает TRUE, если подсказки разрешены, и FALSE в противном случае. При этом параметр bEnable разрешает или запрещает вывод подсказок. Однако только вызова этой функции для разрешения вывода подсказок недостаточно.

Еще одна, скорее вспомогательная, функция *OnToolHitTest*, которую можно перегрузить для настройки ряда параметров, прежде всего, текста будущей подсказки. Нет смысла углубляться в представление этой функции — скажем только, что по умолчанию она:

- подтверждает или опровергает наличие некоторого объекта в точке текущего положения курсора;

- устанавливает флаг, сообщающий системе о необходимости запроса текста для подсказки перед его выводом (LPSTR_TEXTCALLBACK);
- устанавливает флаг центрирования для объектов, которые не являются кнопками (buttons).

У этой функции есть, тем не менее, одно интересное свойство, воспользовавшись которым, можно, по сути, реализовать поддержку всплывающих подсказок еще одним, четвертым способом — для этого следует задать в явном виде строчку текста для вывода подсказки. В этом случае никаких обработчиков сообщений не нужно! Смотрите сами:

```
int CNeptuneWnd::OnToolHitTest(CPoint point, TOOLINFO* pTI) const
{
  // получаем идентификатор элемента управления
  int nHit = CWnd::OnToolHitTest(point, pTI);

  if (pTI != NULL)
  {
    // загружаем текст, определяемый тем же идентификатором,
    // что и элемент управления,
    // но заданный в строковых ресурсах приложения
    CString csText;
    csText.LoadString(nHit);

    if (!csText.IsEmpty())
    {
      pTI->lpszText = new TCHAR[csText.GetLength()+1];
      lstrcpy(pTI->lpszText, (LPCTSTR)csText);
    }
  }

  return nHit;
}
```

Обратите внимание на выделение памяти для хранения текста. Дело в том, что впоследствии код библиотеки MFC самостоятельно освободит ее, в случае, если элемент структуры *TOOLINFO lpszText* не равен LPSTR_TEXTCALLBACK. (При перегрузке функции не забудьте добавить слово "const" после имени и параметров функции как при описании, так и при ее определении.)

Ну и, наконец, поговорим об обработчике сообщения TTN_NEEDTEXT. Такой обработчик нужен как раз в случае, если функция *OnToolHitText* не определена описанным выше способом.

Приведем методику создания обработчика этого сообщения:

- в определении класса функция описывается таким образом:

```
BOOL memberFxn( UINT id, NMHDR * pTTTStruct, LRESULT * pResult);
```

- в карту сообщений добавляется следующая строка:

```
ON_NOTIFY_EX(TTN_NEEDTEXT, 0, memberFxn)
```

- определение функции добавляется следующим образом:

```
BOOL CClassWnd::OnToolTipNotify(UINT id, NMHDR* pNMHDR,
  LRESULT* pResult)
{
  TOOLTIPTEXT *pTTT = (TOOLTIPTEXT *)pNMHDR;

  if (pTTT->uFlags & TTF_IDISHWND)
  {
    // idFrom -фактически дескриптор объекта
    UINT nID = ::GetDlgCtrlID((HWND)pNMHDR->idFrom);

    if (nID)
    {
      pTTT->lpszText = MAKEINTRESOURCE(nID);
      pTTT->hinst = AfxGetResourceHandle();

      return TRUE;
    }
  }
  return FALSE;
}
```

(Более подробно о структуре *TOOLTIPTEXT* смотрите документацию).

Какой бы из двух способов не был выбран, необходимо вызвать функцию *CWnd::EnableToolTips* со значением параметра TRUE.

Панели инструментов с подсказками

Панелей инструментов в окнах, созданных на основе класса *CFrameWnd* и производных от него, стандартно поддерживают подсказки. Причем при генерации каркаса будущего приложения AppWizard делает все необходимое.[1] Для лучшего понимания того, что же все-таки происходит, перечислим действия, которые следует предпринять:

1. Добавление стиля CBRS_TOOLTIPS при создании панели инструментов при помощи функции *CToolBar::Create* или ее настройки функцией *CControlBar::SetBarStyle*.

2. Задание собственно текста подсказок для каждой команды, связанной с кнопками или другими элементами панели. Для этого необходимо доба-

[1] Класс *CToolBarCtrl* обеспечивает доступ к связанному с ним объекту подсказок (функция *GetToolTips*) и позволяет устанавливать новый объект подсказок (функция *SetToolTips*).

вить к описанию команды, которое определено в строковых ресурсах приложения и отображается в строке состояния (если она есть и видна), символа новой строки "\n" и текста подсказки. Кстати, все это можно проделать при редактировании панели инструментов или шаблона меню в редакторе ресурсов. Естественно, что отображаться будут только подсказки, определенные для команд, кнопки которых находятся на панели инструментов.

Более подробно о классах панелей инструментов и других панелей элементов управления см. главу 13 первого тома.

Возвращаемся к классу CToolTipCtrl

Итак, зачем же все-таки нужен класс *CToolTipCtrl*? На самом деле, все способы использования подсказок, рассмотренные ранее базируются как раз на применении этого класса. Вообще, сама библиотека классов MFC тем и хороша, что является как бы надстройкой над Win32 API, а не просто упрощает программирование. Наряду с функциями-членами классов, инкапсулирующих соответствующие объекты Windows, большинство классов библиотеки содержат целый ряд функций, обеспечивающих взаимодействие объектов приложения, связывая их в единое целое, скрывая или упрощая действия, необходимые для программирования под Windows.

Перед описанием класса приведем пример его использования.

```
BOOL CVenusDlg::OnInitDialog()
{
  ...

  // создаем объект-подсказку
  m_ToolTip.Create(this);

  // связываем тексты подсказки с двумя объектами
  m_ToolTip.AddTool(GetDlgItem(IDOK), _T("Подтверждение"));
  m_ToolTip.AddTool(GetDlgItem(IDCANCEL), _T("Отмена"));

  // активизируем (разрешаем) вывод подсказок
  m_ToolTip.Activate(TRUE);
  ...
}

BOOL CVenusDlg::PreTranslateMessage(MSG* pMsg)
{
  m_ToolTip.RelayEvent(pMsg);

  return CDialog::PreTranslateMessage(pMsg);
}
```

Вот и все, что необходимо для обеспечения вывода подсказок для элементов диалога. Обратите внимание, что один и тот же объект класса *CToolTipCtrl* и вместе с ним объект Windows – подсказка, обрабатывает вывод подсказок для более чем одного объекта.

Кроме конструктора, который только создает объект класса, для создания объекта Windows необходимо воспользоваться функцией *Create*:

```
BOOL CToolTip::Create(
  CWnd* pParentWnd,
  DWORD dwStyle = 0) –
```

обязательный параметр pParentWnd должен указывать на окно, элементам которого и будут сопоставлены подсказки. Второй параметр может содержать комбинацию любых стилей элементов управления. Для рассматриваемого класса предусмотрены еще два стиля:

TTS_ALWAYSTIP	обеспечивает вывод текста подсказки независимо от того, активно или нет родительское окно элемента управления
TTS_NOPREFIX	отменяет обработку символа амперсант (&) в качестве управляющего (этот символ указывает на необходимость подчеркивания следующего за ним знака); указание этого стиля, таким образом, позволяет использовать текст пунктов меню для подсказок

Функция *Create* кроме заданных при ее вызове стилей добавляет стили WS_POPUP и WS_EX_TOOLWINDOW.

Основная функция назначения подсказок объектам окна:

```
BOOL CToolTipCtrl::AddTool(
  CWnd* pWnd,
  UINT nIDText,
  LPCRECT lpRectTool = NULL,
  UINT nIDTool = 0)
```

и

```
BOOL CToolTipCtrl::AddTool(
  CWnd* pWnd,
  LPCTSTR lpszText = LPSTR_TEXTCALLBACK,
  LPCRECT lpRectTool = NULL,
  UINT nIDTool=0)
```

Отличие версий заключается в том, каким образом задается текст подсказки. В первом случае это идентификатор, а во втором — указатель на строку текста (или значение LPSTR_TEXTCALLBACK), сообщающий о необходимости запросить текст подсказки перед ее выводом. Для этого следует создать обработчик сообщения TTN_NEEDTEXT родительского окна (см. "Оконная поддержка").

Параметр *pWnd* является указателем на объект класса, связанного с окном Windows, — чаще всего этим объектом является элемент управления. К

сожалению, не всем элементам управления могут быть назначены подсказки. К таким элементам относятся те, которые исключают воздействие на них пользователя, например, текстовые статические элементы (STATIC) или элементы просмотра видео-клипов (ANIMATION).

Если в качестве объекта используются элементы управления, допускающие воздействие со стороны пользователя, то при связывании с ними текста подсказок нет необходимости использовать остальные два параметра функции. С другой стороны, если в качестве первого параметра задан указатель на родительское окно, использование параметров *lpRectTool* и *nIDTool* позволит выводить подсказку в любом месте окна, например, там, где находится статический элемент. Область назначения подсказки задается в координатах относительно левого верхнего угла клиентской области окна, заданного параметром *pWnd*.

```
//
// IDC_TEXT_PROMPT - идентификатор статического
//                   элемента блока диалога
//
CRect rect;
GetDlgItem(IDC_TEXT_PROMPT)->GetWindowRect(&rect);
ScreenToClient(&rect);

// связываем текст подсказки с областью окна там,
// где расположен статический текст
m_ToolTip.AddTool(this, _T("Текст приглашения"),
  &rect, IDC_TECT_PROMPT);
```

Обратная функция *DelTool* отменяет вывод подсказки для заданного указателем окна.

Даже после создания объекта-подсказки сами подсказки выводиться не будут. Для этого необходимо, во-первых, активизировать, другими словами, разрешить вывод подсказок, и, во-вторых, обеспечить поддержку получения соответствующего сообщения о необходимости вывода подсказки объекту подсказки. В примере все это продемонстрировано. Рассмотрим используемые для этого функции.

void CToolTipCtrl::**Activate**(
  BOOL bActivate) –
    разрешает или запрещает в зависимости от значения параметра bActivate вывод подсказок.

Для вывода подсказки необходимо объекту *CToolBarCtrl* передать информацию о местоположении курсора мыши, а также о нажатии клавиш, но уже для того, чтобы спрятать текст подсказки. Для этого в перегруженной функции родительского окна необходимо вызвать функцию *RelayEvent*:

void CToolTipCtrl::**RelayEvent**(LPMSG lpMsg)

которая обрабатывает только следующие сообщения: WM_LBUTTONDOWN, WM_MOUSEMOVE, WM_LBUTTONUP, WM_RBUTTONDOWN, WM_MBUTTONDOWN, WM_RBUTTONUP, WM_MBUTTONUP.

Дополнительно, пожалуй, стоит выделить функцию *HitTest*, позволяющую получить информацию о наличии подсказки по заданным координатам, и функцию *SetDelayTime*, задающую интервал времени, после которого будет выведена подсказка.

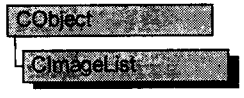
Кроме вышеперечисленных, класс *CToolTipCtrl* имеет в своем составе функции, позволяющие манипулировать подсказками: получить и назначить параметры подсказок (*GetToolInfo* и *SetToolInfo*), получить копию текста подсказки, связанной с конкретным окном (*GetText*), или назначить новый текст (*UpdateTipText*) и некоторые другие.

Глава 25

Комплексные элементы управления

В этом разделе мы рассмотрим общие элементы управления, которые в отличие от рассмотренных в предыдущей главе отличаются более широкими функциональными возможностями. Их использование существенно облегчает создание современных приложений.

Список изображений (класс CImageList)



Список изображений — мощное средство поддержки работы с большим количеством битовых массивов и пиктограмм, доступ к которым осуществляется по номеру, а не по идентификатору. Единственное требование к изображениям списка заключается в том, что все они должны быть одного размера. Все изображения хранятся в одном битовом массиве, в формате, поддерживаемом монитором. Ширина битового массива определяется количеством и шириной хранимых изображений, а их высота определяет высоту массива. Список также может дополнительно хранить монохромный битовый массив, используемый в качестве маски, для реализации "прозрачного" (transparent) вывода. Список поддерживает функции рисования, создания и удаления всего списка, добавления и уничтожения составляющих его элементов, их объединения и переноса (dragging).

Библиотека MFC имеет в своем составе класс *CImageList*, инкапсулирующий возможности списка изображений. Собственно список не является в полном смысле слова элементом управления, т. к. он не является окном, однако при этом список — это объект Windows, и класс *CImageList*, по сути, предоставляет удобный интерфейс с этим объектом. Кроме использования списка для любых нужд разработчика, он в ряде случаев требуется некоторым общим элементам управления — элементу "просмотр списка" (LIST VIEW), элементу "просмотр дерева" (TREE VIEW) и элементу "вкладка" (TAB).

Списки могут быть двух типов: содержащие и несодержащие маски. Списки первого типа содержат два битовых массива одного размера, один — для хранения собственно изображений, другой — для хранения монохромного битового массива, который содержит маски для каждого изображения.

Создание списка

Создание списка при помощи класса *CImageList* осуществляется в два этапа. Во-первых, следует создать объект класса, а затем вызвать одну из версий функции *Create* для создания списка-объекта Windows. Для хранения дескриптора последнего класс имеет в своем составе член *m_hImageList*.

HIMAGELIST CImageList::**m_hImageList**

доступ к которому при необходимости рекомендуется осуществлять при помощи функций GetSafeHandle, Attach и Detach.

Функции создания существенно отличаются по производимым ими действиям. Рассмотрим их подробнее.

```
BOOL CImageList::Create(
  int cx,
  int cy,
  BOOL bMask,
  int nInitial,
  int nGrow) —
```

создает пустой список с заданными при помощи параметров *cx* и *cy* размерами битовых массивов, которые в дальнейшем можно будет добавить в список. При задании ненулевого значения параметра nInitial, сразу при создании списка создается битовый массив, который в дальнейшем будет содержать изображения. Параметр nGrow позволяет более эффективно выделять память для хранения изображений, он задает число, которое определяет, на сколько изображений следует увеличивать битовый массив, при необходимости. Параметр bMask определяет, какого типа следует создавать список — содержащий маски или нет. Все остальные версии функции создают списки, в обязательном порядке содержащие маски.

Следующие две версии функции *Create* позволяют создавать уже заполненные списки, содержимое которых, впрочем, можно изменять во время работы:

```
BOOL CImageList::Create(
  UINT nBitmapID,
  int cx,
  int nGrow,
  COLORREF crMask)
```

и

```
BOOL CImageList::Create(
  LPCTSTR lpszBitmapID,
  int cx,
  int nGrow,
  COLORREF crMask).
```

Параметры nBitmapID и lpszBitmapID задают дескрипторы битовых массивов, уже содержащих изображения, количество которых определяется их шириной (параметр cx). Параметр crMask позволяет указать цвет для создания масок и реализации прозрачного вывода изображений. Цвет каждой точки изображения, совпадающий с цветом, заданным параметром crMask, заменяется на черный, и устанавливается соответствующий бит маски.

Последняя версия функции позволяет создать список на основе двух других:

```
BOOL CImageList::Create(
   CImageList& imagelist1,
   int nImage1,
   CImageList& imagelist2,
   int nImage2,
   int dx,
   int dy) —
```

объединяет содержимое двух списков таким образом, что изображения второго списка накладываются на изображения первого, а маска создается в результате выполнения логической операции OR для каждого бита масок обоих списков. Параметры dx и dy определяют размеры изображений.

Изменение содержимого списка

Для добавления, удаления и замены изображений используются функции *Add*, *Remove* и *Replace* соответственно.

```
int CImageList::Add(
   CBitmap* pbmImage,
   CBitmap* pbmMask)

int CImageList::Add(
   CBitmap* pbmImage,
   COLORREF crMask)

int CImageList::Add(HICON hIcon) —
```

позволяют не только добавлять сразу несколько битовых массивов, количество которых определяется заданной при создании шириной изображений списка, но и задавать маску для изображений. Последняя версия функции позволяет добавлять в список пиктограмму, которая также содержит маску.

```
BOOL CImageList::Replace(
   int nImage,
   CBitmap* pbmImage,
   CBitmap* pbmMask)

int CImageList::Replace(
   int nImage,
   HICON hIcon) —
```

позволяют заменить изображение по его номеру как битовым массивом (параметр pbmImage), так и пиктограммой.

Пиктограммы, в отличие от битовых массивов, уже содержат маску, позволяющую производить рисование в прозрачном режиме. Следующая функция создает пиктограмму на основе битового массива и маски (если она есть):

```
HICON CImageList::ExtractIcon(int nImage) —
```

возвращает дескриптор созданной пиктограммы для изображения по номеру, задаваемому параметром nImage.

Для заполнения и сохранения всех изображений списка реализованы функции *Read* и *Write*:

BOOL CImageList::**Read**(CArchive* pArchive)

и

BOOL CImageList::**Write**(CArchive* pArchive) —

используют указатели на объекты класса CArchive. Эти объекты следует создать самостоятельно.

Так как объект класса *CImageList* лишь представляет в программе объект Windows, путем связывания с ним последнего можно обеспечить доступ к ранее созданному другими средствами списку. К функциям данной группы относятся функции *Attach* и *Detach*.

Рисование изображений

Для отображения в прозрачном режиме, когда все точки области устройства вывода изображения имеют один и тот же цвет, следует задать цвет фона изображения таким же, что и цвет области. В этом случае существенно ускоряется процесс вывода, т. к. не требуется дополнительное использование маски. Для этого класс *CImageList* имеет в своем составе функцию:

COLORREF CImageList::**SetBkColor**(COLORREF cr)

которая возвращает ранее установленный цвет фона. По умолчанию цвет имеет значение CLR_NONE, в этом случае прозрачный режим реализуется с использованием маски.

Для рисования изображений следует использовать функцию *Draw*:

```
BOOL CImageList::Draw(
   CDC* pdc,
   int nImage,
   POINT pt,
   UINT nStyle)
```

которая осуществляет вывод на устройство, определяемое указателем (параметр pdc) на объект класса CDC, по координатам, задаваемым параметром pt. Изображение определяется индексом (параметр nImage). Параметр nStyle задает режим вывода изображения и может содержать комбинацию следующих значений.

ILD_NORMAL	рисование осуществляется на основе цвета, заданного вызовом функции *SetBkColor*, что позволяет реализовать прозрачный режим и при отсутствии маски
ILD_TRANSPARENT	рисование осуществляется с использованием маски вместо цвета фона
ILD_BLEND50	при наличии маски происходит вывод изображения, "подсвеченного" цветом выделения, для индикации выбора, режим игнорируется при отсутствии маски

ILD_BLEND25 при наличии маски происходит вывод изображения, заштрихованного с использованием цвета выделения, для индикации наличия фокуса

ILD_OVERLAYMASK происходит вывод изображения с участием маски перекрытия; совместно с флагом следует указать индекс маски перекрытия,[1] заданного при помощи макроса INDEXTOOVERLAYMASK

Перемещение изображений

Как ни странно, класс списка изображений поддерживает набор функций-членов для осуществления процесса "перетаскивания" его элементов. Достоинство реализации в том, что перемещение изображения происходит плавно, изображение при этом остается полноцветным и исключается мигание курсора мыши.

Для того чтобы начать процесс перемещения изображения, следует вызвать функцию *BeginDrag*:

BOOL CImageList::**BeginDrag**(
  int nImage,
  CPoint ptHotSpot)

параметр nImage задает порядковый номер изображения, а параметр ptHotSpot — координаты точки привязки (hot spot) на изображении, которая и будет перемещаться, и относительно нее будет производиться отображение изображения.

Совместно с функцией *BeginDrag* следует вызвать функцию:

static BOOL CImageList::**DragEnter**(
  CWnd* pWndLock,
  CPoint point)

которая блокирует перерисовку окна в процессе перемещения изображения. Первый параметр указывает на окно, изменения в котором будут заблокированы. Значение этого параметра, равное NULL, позволяет перемещать изображение и вне окна, по всему экрану, при этом окном, где будет происходить перемещение, будет являться "рабочий стол" (desktop window). Второй параметр задает точку в окне, которой будет соответствовать точки привязки. Координаты точки задаются в области всего окна, а не только в его клиентской части, следовательно, требуется произвести соответствующие преобразования.

В процессе перемещения (обычно в обработчике сообщения WM_MOUSEMOVE) для отображения текущего положения изображения следует пользоваться функцией *DragMove*:

static BOOL CImageList::**DragMove**(CPoint pt)

которая принимает координаты нового положения точки привязки также в области всего окна.

---

[1] Каждый список изображений может иметь до четырех масок перекрытия, задаваемых функцией *SetOverlayImage*. В качестве масок перекрытия используются изображения, содержащиеся в списке.

По завершении операции перемещения следует сообщить системе о необходимости разблокирования перерисовки окна (функция *DragLeave*) и завершить собственно перемещение (функция *EndDrag*).

Если возникает необходимость рисования в окне во время перемещения изображения, следует либо по мере необходимости разблокировать окно при помощи функции *DragLeave* (с последующей повторной блокировкой), либо использовать функцию *DragShowNolock*.

Кроме вышеперечисленных, полезной может оказаться функция *SetDragCursorImage* для временного изменения изображения на момент перемещения путем комбинирования перемещаемого изображения с некоторым другим, например, с изображением курсора.

Далее приведем пример программы (приложение ImageDrag), в которой реализован описанный выше механизм — из начального положения изображение при помощи мыши может быть перенесено в любую часть окна.

```
/*************************************************************
Файл ImageDragFrm.cpp
Copyright (c) 1997 Мешков А., Тихомиров Ю., Шведов Д.
*************************************************************/

// ImageDragFrm.cpp : implementation file
//

#include "stdafx.h"
#include "ImageDrag.h"
#include "ImageDragFrm.h"

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

/////////////////////////////////////////////////////////////
// CImageDragFrame

CImageDragFrame::CImageDragFrame()
{
}

CImageDragFrame::~CImageDragFrame()
{
}

BEGIN_MESSAGE_MAP(CImageDragFrame, CFrameWnd)
  //{{AFX_MSG_MAP(CImageDragFrame)
  ON_WM_PAINT()
  ON_WM_CREATE()
  ON_WM_LBUTTONDOWN()
  ON_WM_MOUSEMOVE()
```

```
  ON_WM_LBUTTONUP()
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////
// CImageDragFrame message handlers

void CImageDragFrame::OnPaint()
{
  CPaintDC dc(this);

  // отображение изображения
  m_List.Draw(&dc, 0, m_pt, ILD_TRANSPARENT);
}

int CImageDragFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
  if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
    return -1;

  // создание списка изображений
  m_List.Create(32, 32, TRUE, 1, 1);
  // добавление пиктограммы
  m_List.Add(AfxGetApp()->LoadIcon(IDR_IMAGEDRAG));

  // начальная инициализация дополнительных параметров
  m_pt = CPoint(0, 0);
  m_Drag = FALSE;

  return 0;
}

void CImageDragFrame::OnLButtonDown(UINT nFlags, CPoint point)
{
  CRect rect(m_pt.x, m_pt.y, m_pt.x+32, m_pt.y+32);

  // проверка на наличие курсора мыши в рамках пиктограммы
  if (rect.PtInRect(point))
  {
    // начало перемещения
    m_Drag = TRUE;
    m_List.BeginDrag(0, CPoint(0, 0));

    // преобразование координат
    GetWindowRect(&rect);

    CPoint pt = point;
    ClientToScreen(&pt);
    m_List.DragEnter(this,
      CPoint(pt.x - rect.left, pt.y - rect.top));
  }
  else
    m_Drag = FALSE;
```

```
  CFrameWnd::OnLButtonDown(nFlags, point);
}

void CImageDragFrame::OnMouseMove(UINT nFlags, CPoint point)
{
  if (m_Drag)
  {
    // преобразование координат
    CRect rect;
    GetWindowRect(&rect);

    CPoint pt = m_pt = point;
    ClientToScreen(&pt);

    // перемещение
    m_List.DragMove(CPoint(pt.x - rect.left, pt.y - rect.top));
  }

  CFrameWnd::OnMouseMove(nFlags, point);
}

void CImageDragFrame::OnLButtonUp(UINT nFlags, CPoint point)
{
  if (m_Drag)
  {
    // завершение перемещения
    m_List.DragLeave(this);

    m_List.EndDrag();
    m_Drag = FALSE;

    // инициируем перерисовку окна, т. к. после вызова
    // функции DragLeave изображение было скрыто
    InvalidateRect(NULL);
  }

  CFrameWnd::OnLButtonUp(nFlags, point);
}
```

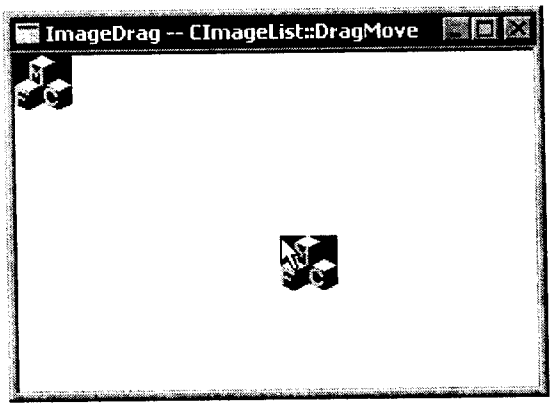


Рис. 26. Окно приложения ImageDrag в момент перемещения изображения

Информация о списке

Для получения информации о параметрах списка следует пользоваться функциями *GetImageCount*, возвращающей количество элементов (изображений) списка, и *GetImageInfo*, заполняющей структуру *IMAGEINFO* информацией об изображении по его порядковому номеру в списке:

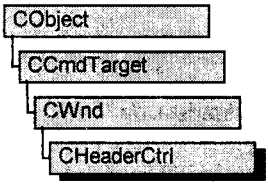
```
BOOL CImageList::GetImageInfo(
  int nImage,
  IMAGEINFO* pImageInfo)
```

Для получения информации об изображении, следует передать указатель на структуру *IMAGEINFO*:

```
typedef struct _IMAGEINFO {
  HBITMAP hbmImage;
  HBITMAP hbmMask;
  int cPlanes;
  int cBitsPerPixel;
  RECT rcImage;
} IMAGEINFO;
```

Первые два поля используются для хранения дескрипторов битового массива и маски, поля *cPlanes* и *cBitsPerPixel* содержат количество плоскостей и битов, требуемых для хранения цвета соответственно, а поле *rcImage* — размеры битового массива.

Заголовок (класс CHeaderCtrl)



В этом разделе мы рассмотрим еще один элемент управления — "заголовок" и соответствующий ему класс библиотеки MFC *CHeaderCtrl*.

Заголовок полезен для использования вместе со списками, создаваемыми с помощью класса *CListCtrl*, но это не значит, что его нельзя использовать в других случаях, например, при создании электронных таблиц. Заголовок обычно разбит на части, называемые элементами заголовка. Каждая из них является собственно заголовком для соответствующего столбца. Разделители можно передвигать, меняя ширину столбцов, в случае, если элемент заголовка создан с соответствующим стилем.

На рис. 27 показано, как данный элемент управления может быть использован для создания окна, в котором выводится информация о файлах на диске.

Можно создать элемент заголовка, по внешнему виду похожий и действующий при щелчке на нем мышью так же, как кнопки, срабатывающие по

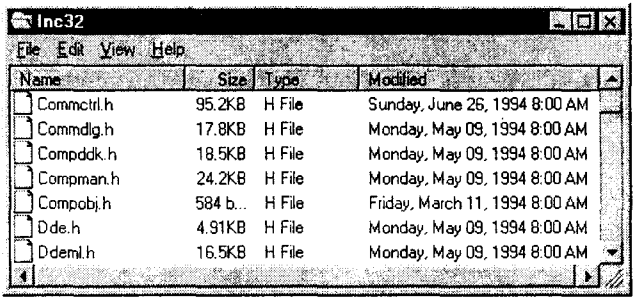


Рис. 27. Использование элемента управления "заголовок"

нажатию. Для этого нужно задать стиль HDS_BUTTONS. В этом случае можно реализовать, например, сортировку по алфавиту столбца записей, осуществляемую при нажатии на элемент заголовка соответствующего столбца.

Каждый элемент заголовка может содержать текстовую строку или битовый массив и 32-битное значение, ассоциированное с ним, которое определяется приложением.

Создание элемента заголовка

Элемент заголовка невозможно непосредственно создать в редакторе ресурсов, более того, и средство ClassWizard не поддерживает добавление членов класса *CHeaderCtrl*.

Для того чтобы вставить заголовок в блок диалога, следует:

1. вручную создать объект класса *CHeaderCtrl* в классе блока диалога;
2. в функции *OnInitDialog* создать собственно объект-заголовок, используя функцию *Create*;
3. добавить в заголовок элементы заголовка;
4. с помощью средства ClassWizard добавить необходимые обработчики сообщений в классе блока диалога.

Для того чтобы вставить заголовок в окно просмотра, не базирующееся на классе *CListView*, следует:

1. вручную создать объект класса *CHeaderCtrl* в классе представления;
2. в функции *OnInitialUpdate* создать собственно объект-заголовок, используя функцию *Create*;
3. добавить в заголовок элементы заголовка;
4. с помощью средства ClassWizard добавить необходимые обработчики сообщений в класс представления.

Как вы можете видеть, отсутствие поддержки со стороны средства разработки приложений приводит к тому, что создание заголовка и в блоках диалога

и в окнах-представлениях практически идентично. В обоих случаях встроенный объект создается тогда, когда создается блок диалога или окно-представление при помощи вызова функции *Create*:

```
BOOL CHeaderCtrl::Create(
   DWORD dwStyle,
   const RECT& rect,
   CWnd* pParentWnd,
   UINT nID)
```

Параметр *dwStyle* может содержать любые комбинации стилей элементов управления, включая следующие, свойственные для рассматриваемого элемента управления:

HDS_BUTTONS	элементы заголовка выглядят как кнопки, срабатывающие по нажатию
HDS_HORZ	заголовок имеет горизонтальную ориентацию
HDS_VERT	заголовок имеет вертикальную ориентацию (к сожалению, в настоящий момент этот стиль не поддерживается)
HDS_HIDDEN	изначально заголовок не виден

Следующая группа стилей определяет, где элемент заголовка располагается и как изменяет свои размеры:

CCS_BOTTOM	заголовок располагается в нижней части клиентской области родительского окна и имеет ширину, равную ширине родительского окна
CCS_NOMOVEY	заставляет заголовок в ответ на сообщение WM_SIZE изменять размер и перемещаться по горизонтали, но не по вертикали; если уже использован стиль CCS_NORESIZE, то этот стиль не применим; элемент заголовка по умолчанию имеет данный стиль
CCS_NOPARENTALIGN	предотвращает заголовок от автоматического перемещения в верхнюю или нижнюю часть родительского окна; элемент заголовка сохраняет свое расположение внутри родительского окна, несмотря на изменение его размеров; если использованы стили CCS_TOP или CCS_BOTTOM, то высота устанавливается по умолчанию, а расположение и ширина не изменяются
CCS_NORESIZE	заголовок не использует ширину и высоту, заданные по умолчанию; он должен использовать данные, заданные при создании или изменении размеров
CCS_TOP	заголовок располагается в верхней части клиентской области родительского окна и имеет ширину, равную ширине родительского окна

Параметр *rect* задает размер и расположение заголовка. Параметр *pParentWnd* задает родительское окно. Значение этого параметра не должно быть равно NULL. Параметр *nID* задает идентификатор заголовка.

Изменение элементов заголовка

После того как заголовок создан, можно добавлять в него элементы. Обычно устанавливают один элемент на столбец. Для этого нужно выполнить следующие шаги для каждого добавляемого элемента заголовка:

1. подготовить структуру *HD_ITEM*;
2. вызвать функцию *InsertItem*, передав ей указатель на заполненную структуру *HD_ITEM*.

Структура *HD_ITEM* имеет следующий вид:

```
typedef struct _HD_ITEM {
    UINT mask;
    int cxy;
    LPSTR pszText;
    HBITMAP hbm;
    int cchTextMax;
    int fmt;
    LPARAM lParam;
} HD_ITEM;
```

Поля имеют следующие назначения: *cxy* определяет ширину или высоту элемента в зависимости от ориентации заголовка, поле *pszText* является указателем на тестовую строку, длина которой определяется значением поля *cchTextMax*, поле *hbm* является дескриптором битового массива, поле *lParam* задает 32-битное значение для элемента заголовка. Поле *fmt* задает способ форматирования содержимого элемента заголовка и может содержать комбинацию следующих значений:

HDF_CENTER	содержимое элемента выравнивается по центру
HDF_LEFT	содержимое элемента выравнивается по левому краю
HDF_RIGHT	содержимое элемента выравнивается по правому краю
HDF_BITMAP	элемент содержит битовый массив
HDF_OWNERDRAW	рисование элемента осуществляется в коде, написанном разработчиком приложения
HDF_STRING	элемент содержит текстовую строку

Чтобы определять, какие поля структуры содержат достоверные значения для осуществления вывода, структура имеет поле *mask*, которое может содержать комбинацию следующих значений:

HDI_BITMAP	задано поле *hbm*
HDI_FORMAT	задано поле *fmt*
HDI_HEIGHT	задано поле *cxy*, и оно содержит высоту элемента
HDI_LPARAM	задано поле *lParam*

HDI_TEXT заданы поля *pszText* и *cchTextMax*

HDI_WIDTH задано поле *cxy*, и оно содержит ширину элемента

Итак, для того чтобы добавить новый элемент в заголовок по заданному индексу, предназначена функция *InsertItem*:

```
int CHeaderCtrl::InsertItem(
   int nPos,
   HD_ITEM* phdi) —
```

параметр nPos определяет индекс элемента, перед которым будет осуществлена вставка. Если значение параметра больше текущего количества элементов заголовка, то новый элемент будет добавлен в конец заголовка.

Следующая пара функций позволяет получить и изменить атрибуты элементов заголовка.

```
BOOL CHeaderCtrl::GetItem(
   int nPos,
   HD_ITEM*pHeaderItem)
```

и

```
BOOL CHeaderCtrl::SetItem(
   int nPos,
   HD_ITEM *pHeaderItem)
```

Для получения информации о конкретном элементе, также следует задавать поле *mask* структуры HD_ITEM, в противном случае информация возвращена не будет.

Для удаления элемента из заголовка предназначена функция *DeleteItem*. А для получения общего количества элементов заголовка — *GetItemCount*.

Обработка извещений элемента заголовка

Для осуществления управления заголовком и связанными с ним столбцами и строками, с помощью средства ClassWizard необходимо добавить в класс родительского окна или блока диалога обработчики сообщений для функции *OnNotify*. Сообщения посылаются в рамках сообщения WM_NOTIFY родительскому окну. Перечислим уведомляющие сообщения, которые может посылать заголовок:

HDN_ITEMCLICK пользователь щелкнул мышью на элементе заголовка

HDN_ITEMCHANGED атрибуты элемента заголовка изменились

HDN_ITEMCHANGING атрибуты элемента заголовка изменяются

HDN_ITEMDBLCLICK пользователь произвел двойной щелчок мышью на элементе заголовка

HDN_DIVIDERDBLCLICK пользователь произвел двойной щелчок мышью на разделителе

HDN_BEGINTRACK	пользователь начал перемещать разделитель
HDN_TRACK	пользователь перемещает разделитель
HDN_ENDTRACK	пользователь закончил перемещать разделитель

Для получения информации о конкретном элементе, в результате взаимодействия пользователя с которым произошла генерация уведомления, следует преобразовать параметр *lParam* к указателю на структуру *HD_NOTIFY*. Эта структура содержит следующую информацию об элементе:

```
typedef struct _HD_NOTIFY {
  NMHDR hdr;
  int iItem;
  int iButton;
  HD_ITEM FAR* pitem;
} HD_NOTIFY;
```

Поле *iItem* содержит индекс элемента (поле *iButton*), который, возможно, был нажат (0 — левая кнопка, 1 — правая, 2 — средняя), поле *pitem* является указателем на ранее описанную структуру *HD_ITEM*.

Чтобы проиллюстрировать вышесказанное, приведем пример использования заголовков (приложение Header).

В функции *OnInitialUpdate* класса блока диалога *CHeaderDlg* создается заголовок из пяти элементов одинаковой ширины.

```
/************************************************************
Файл HeaderDlg.cpp
Copyright (c) 1997 Мешков А., Тихомиров Ю., Шведов Д.
************************************************************/
...
BOOL CHeaderDlg::OnInitDialog()
{
  ...
  CRect rect;
  GetClientRect(&rect);
  rect.bottom = 30;
  rect.DeflateRect(5, 5);
  m_Horz.Create(HDS_BUTTONS | HDS_HORZ | CCS_TOP |
    WS_CHILD | WS_VISIBLE, rect, this, 1000);
  int nWidth = rect.Width() / 5;
  TCHAR szText[2];
  HD_ITEM item = { HDI_TEXT | HDI_WIDTH, nWidth, szText, NULL, 1 };
  for (int i = 0; i < 5; i++)
  {
    m_Offset[i] = nWidth;
```

```
    szText[0] = 'A'+i; szText[1] = '\0';
    m_Horz.InsertItem(i, &item);
  }
  ...
}
```

В функции *OnPaint* в зависимости от ширины элементов заголовка рисуется сетка, в пять столбцов и несколько строк.

```
void CHeaderDlg::OnPaint()
{
  if (IsIconic())
  {
    ...
  }
  else
  {
    CPaintDC dc(this); // device context for painting

    CPen pen(PS_DOT, 1, RGB(0, 0, 0));
    CPen *pPen = dc.SelectObject(&pen);

    CRect rect;
    GetClientRect(&rect);
    rect.DeflateRect(5, 5);
    rect.top += 20;

    dc.FillSolidRect(&rect, RGB(255, 255, 255));

    int nOffset = 4;

    for (int i = 0; i < 5; i++)
    {
      if ((nOffset += m_Offset[i]) >= rect.right)
        break;

      dc.MoveTo(nOffset, 20);
      dc.LineTo(nOffset, rect.bottom);
    }

    for (i = 1; i < 10; i++)
    {
      dc.MoveTo(rect.left, rect.top + (rect.Height() / 10) * i);
      dc.LineTo(rect.right, rect.top + (rect.Height() / 10) * i);
    }

    dc.SelectObject(pPen);

    CDialog::OnPaint();
  }
}
```

Функция осуществляет обработку сообщения, уведомляющего об окончании изменения ширины элемента заголовка, и инициирует перерисовку содержимого окна.

```
BOOL CHeaderDlg::OnNotify(WPARAM wParam, LPARAM lParam,
  LRESULT* pResult)
{
  NMHDR *nmhdr = (NMHDR *)lParam;

  if (nmhdr->hwndFrom == m_Horz.m_hWnd)
  {
    if (nmhdr->code == HDN_ENDTRACK)
    {
      HD_NOTIFY *phdn = (HD_NOTIFY *)lParam;
      m_Offset[phdn->iItem] = phdn->pitem->cxy;

      InvalidateRect(NULL);
    }
  }
```

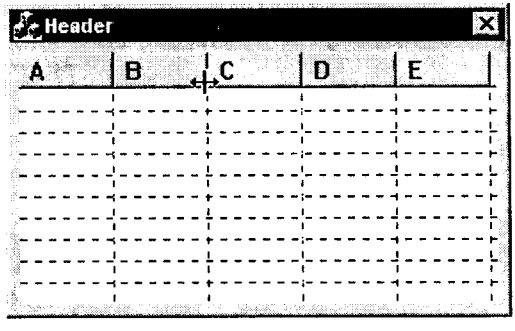


Рис. 28. Окно приложения Header

Элемент заголовка, внешний вид которого определяется пользователем

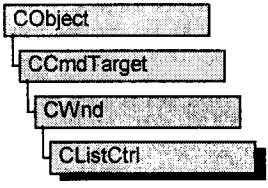
Можно определить вывод отдельных элементов заголовка, внешний вид которых определяется пользователем. Для этого при создании элемента нужно, чтобы поле *fmt* структуры *HD_ITEM* содержало флаг HDF_OWNERDRAW. Также следует создать класс на базе класса *CHeaderCtrl* и переопределить функцию *DrawItem*. Эта функция будет вызываться всякий раз, когда возникнет необходимость перерисовки элемента (элементов, если их несколько) заголовка:

void CHeaderCtrl::**DrawItem**(LPDRAWITEMSTRUCT lpDrawItemStruct)

Ранее мы уже рассматривали структуру *DRAWITEMSTRUCT*, однако в случае заголовка ее поля имеют несколько иное значение. Поле *CtlType* содер-

жит значение ODT_HEADER, поле *CtlID* — идентификатор окна заголовка, поле *itemID* — порядковый номер элемента заголовка, для рисования которого вызывается функция *DrawItem*, поле *itemAction* всегда содержит значение ODA_DRAWENTIRE, т. к. требуется перерисовка всего элемента, поле *itemState* — значение ODS_SELECTED, если курсор мыши находится на элементе и этот элемент находится в нажатом состоянии, в противном случае значение поля равно 0, поля *hwndItem* и *hDC* содержат дескрипторы окна заголовка и контекста устройства соответственно, поле *rcItem* определяет границы элемента заголовка и поле *itemData* содержит 32-битное число, ассоциированное с элементом.

Просмотр списка (класс CListCtrl)



Просмотр списка — мощное средство представления упорядоченного списка элементов, каждый из которых может иметь структуру, состоящую из нескольких полей. С каждой записью могут быть ассоциированы до трех пиктограмм различного размера, отображение которых может нести дополнительную визуальную информацию.

Режимы вывода

Элемент "просмотра списка" (список просмотра) может выводить записи четырьмя различными способами, наличие которых существенно отличает этот элемент управления от всех других. При этом возможно переключение от одного режима представления к другому после создания элемента управления. Рассмотрим режимы вывода подробнее:

Режим вывода	Описание
Режим больших пиктограмм (Icon view)[1]	задается стилем LVS_ICON; каждая запись представляет собой пиктограмму, под которой расположена надпись; записи могут перемещаться пользователем внутри списка просмотра
Режим малых пиктограмм (Small icon)[2]	задается стилем LVS_SMALLICON; каждая запись выводится в виде маленькой пиктограммы с надписью, расположенной справа от нее; записи могут перемещаться пользователем внутри списка

[1] Размер больших пиктограмм — 32 × 32 пикселя.

[2] Размер малых пиктограмм — 16 × 16 пикселей.

(продолжение)

Режим вывода	Описание
Режим списка (List view)	задается стилем LVS_LIST; каждая запись выводится в виде маленькой пиктограммы с надписью, расположенной справа от нее; в отличие от предыдущего способа вывода, записи выводятся столбцами и пользователь не может менять их расположение внутри списка
Режим таблицы (Report view)	задается стилем LVS_REPORT; каждая запись с информацией о ней выводится в отдельной строке; пиктограмма и надпись выводятся в столбцах, расположенных слева, а в столбцах справа приложение может размещать дополнительную информацию для каждой записи; если не задан стиль LVS_NOCOLUMNHEADER, то каждый столбец имеет заголовок

Примером применения просмотра списка может служить правая часть окна программы Windows Explorer, которая позволяет установить перечисленные выше в таблице способы вывода записей в окне.

Для задания выравнивания пиктограмм и надписей при первых двух способах вывода можно применять стили LVS_ALIGNTOP — выравнивание по верхнему краю (по умолчанию) или LVS_ALIGNLEFT — выравнивание по левому краю. Способ выравнивания также можно изменять уже после создания просмотра списка. Для отделения стилей, задающих выравнивание записей можно применять стиль LVS_ALIGNMASK.

Дополнительные стили окна позволяют использовать другие возможности, например, редактировать надписи под пиктограммами непосредственно в просмотре списка, выбирать больше одной записи и т. д.

Создание просмотра списка

Библиотека MFC имеет в своем составе класс, инкапсулирующий возможности рассматриваемого элемента управления. Это класс *CListCtrl*, его использование существенно упрощает и использование элемента управления. Как и большинство других элементов управления, просмотр списка создается или в блоке диалога добавлением в шаблон блока диалога соответствующего элемента или в обычном окне при помощи вызова функции *Create* в функциях окна *OnInitDialog* (для диалогов) или *OnCreate*:

```
BOOL CListCtrl::Create(
   DWORD dwStyle,
   const RECT& rect,
   CWnd* pParentWnd,
   UINT nID)
```

Параметр *dwStyle* может содержать любые комбинации стилей элементов управления, а также следующие, свойственные для рассматриваемого элемента управления:

LVS_ALIGNLEFT	записи выравниваются по левому краю при просмотре в режиме больших или малых пиктограмм
LVS_ALIGNTOP	записи выравниваются по верхнему краю при просмотре в режиме больших или малых пиктограмм
LVS_AUTOARRANGE	пиктограммы автоматически выравниваются при просмотре в режиме больших или малых пиктограмм при добавлении, удалении или перемещении записей
LVS_EDITLABELS	позволяет редактировать текстовые надписи непосредственно в списке просмотра, для поддержки этого режима необходимо обрабатывать уведомляющее сообщение LVN_ENDLABELEDIT
LVS_ICON	задает режим больших пиктограмм
LVS_LIST	задает режим списка
LVS_NOCOLUMNHEADER	заголовки столбцов не выводятся на экран в случае просмотра в режиме таблицы; по умолчанию заголовки выводятся на экран
LVS_NOLABELWRAP	текст в надписях выводится в виде одной строки без переноса на следующую; по умолчанию текст может быть перемещен в случае просмотра в режиме больших и малых пиктограмм
LVS_NOSCROLL	отключает прокрутку списка просмотра в окне; все записи должны вмещаться в клиентскую область окна
LVS_NOSORTHEADER	отключает стиль имитации кнопок для элементов заголовка
LVS_OWNERDRAWFIXED	требует написания кода для реализации отображения записей в случае просмотра в виде таблицы; список просмотра посылает сообщение WM_DRAWITEM в случае необходимости перерисовки каждой записи, но не посылает отдельные сообщения для подзаписей (см. ниже)
LVS_REPORT	задает режим таблицы, для возможности отображения пиктограмм совместно с записями таблицы следует комбинировать с этим стилем стиль LVS_ICON
LVS_SHAREIMAGELISTS	список просмотра использует список изображений совместно с другими элементами управления
LVS_SINGLESEL	позволяет в данный момент времени выбирать только одну запись в списке просмотра
LVS_SMALLICON	задает режим малых пиктограмм
LVS_SORTASCENDING	сортирует текстовые записи по возрастанию
LVS_SORTDESCENDING	сортирует текстовые записи по убыванию

Параметр *rect* задает размер и расположение списка просмотра. Параметр *pParentWnd* задает родительское окно. Значение этого параметра не должно быть равно NULL. Параметр *nID* задает идентификатор списка просмотра.

Параметры просмотра списка

Как уже упоминалось, стиль просмотра списка можно менять в любое время уже после его создания. Для этого, к сожалению, библиотекой классов MFC не предусмотрены соответствующие функции, поэтому следует использовать функции *::GetWindowLong* и *::SetWindowLong*. В следующем примере приведена функция, позволяющая это делать.

```
void CSaturnListCtrl::SetStyle(DWORD dwViewStyle)
{
  ASSERT(::IsWindow(m_hWnd));

  // получаем текущие стили окна
  DWORD dwStyle = GetWindowLong(m_hWnd, GWL_STYLE);

  // устанавливаем стиль, если только он не был задан ранее
  if ((dwStyle & LVS_TYPEMASK) != dwViewStyle)
    SetWindowLong(m_hWnd, GWL_STYLE,
    (dwStyle & ~LVS_TYPEMASK) | dwViewStyle);
}
```

Класс *CListCtrl* предусматривает возможность установки и изменения цвета фона списка просмотра. Следующая пара функций предназначена для получения значения текущего цвета фона (*GetBkColor*) и установки нового (*SetBkColor*). Если значение текущего цвета равно CLR_NONE, т. е. цвет фона не задан, перерисовка списка просмотра осуществляется значительно медленнее, чем при задании цвета.

Цвет фона — не единственный цвет, значение которого можно получить или изменить. Функции *GetTextColor* и *SetTextColor* используются для работы с цветом букв текста, а функции *GetTextBkColor* и *SetTextBkColor* — для работы с цветом фона текстовых строк.

Функции для работы со списком в целом

Следующие функции предназначены для управления выводом части записей списка в случае, если все они не могут быть расположены в окне просмотра списка.

BOOL CListCtrl::**GetViewRect**(LPRECT lpRect) —
возвращает размеры прямоугольника, ограничивающего все записи списка в режиме просмотра с малыми или большими пиктограммами.

int CListCtrl::**GetTopIndex**() —
возвращает индекс верхней видимой записи.

int CListCtrl::**GetCountPerPage**() —
возвращает количество записей, которые могут поместиться по вертикали в видимой части просмотра списка при просмотре в режиме списка или таблицы.

BOOL CListCtrl::**GetOrigin**(LPPOINT lpPoint) —
возвращает координаты, с которых начинается вывод элементов просмотра списка .

BOOL CListCtrl::**EnsureVisible**(
int nItem,
BOOL bPartialOK) —
гарантирует, что запись, заданная индексом[1] (параметр nItem), будет видимой. Достигается это путем прокрутки. Параметр bPartialOK определяет допустимость отображения только части элемента. Если значение параметра равно TRUE, то прокрутка не производится, если запись уже частично видна.

BOOL CListCtrl::**Scroll**(CSize size) —
функция осуществляет прокрутку по вертикали и горизонтали на заданное количество пикселей.

BOOL CListCtrl::**RedrawItems**(
int nFirst,
int nLast) —
заставляет просмотр списка перерисовать записи с индексами, лежащими в диапазоне от nFirst и до nLast включительно. Однако указанная группа записей не перерисовывается немедленно — это происходит по получении списком сообщения WM_PAINT. Для немедленной перерисовки конкретной записи предназначена функция UpdateWindow.

BOOL CListCtrl::**Update**(int nItem) —
перерисовывает заданную запись просмотра списка. При этом, возможно, производится переупорядочивание записей (в случае, если просмотр списка имеет стиль LVS_AUTOARRANGE).

BOOL CListCtrl::**Arrange**(UINT nCode) —
эта функция инициирует переупорядочивание записей. Способ упорядочивания задается параметром nCode, который может принимать одно из следующих значений:

LVA_ALIGNLEFT	записи должны быть выровнены по левому краю окна
LVA_ALIGNTOP	записи должны быть выровнены по верхнему краю окна
LVA_DEFAULT	записи должны быть переупорядочены в соответствии с текущим установленным типом выравнивания
LVA_SNAPTOGRID	пиктограммы должны быть помещены в ближайшую точку сетки для выравнивания

[1] См. "Реализация просмотра списка с возможностью перемещения записей".

Также указанные значения могут комбинироваться со следующими значениями:

LVA_SORTASCENDING	сортировка записей в порядке возрастания
LVA_SORTDESCENDING	сортировка записей в порядке убывания

Списки изображений

Пиктограммы для записей просмотра списка содержатся в списках изображений, которые необходимо создать и определить, если есть необходимость сопровождать записи пиктограммами. Один список изображений содержит полноразмерную пиктограмму (для режима больших пиктограмм), второй — пиктограммы меньшего размера (для остальных режимов). Можно определить и третий список, который будет отображать определяемые приложением состояния записей.

Для работы со списком изображений предназначены две функции:

```
CImageList* CListCtrl::GetImageList(int nImageList)
и
CImageList* CListCtrl::SetImageList(
   CImageList* pImageList,
   int nImageList)
```

Первая возвращает указатель на текущий список изображений, а вторая устанавливает текущий новый список изображений. Параметр *nImageList* в обеих функциях может принимать одно из следующих значений:

LVSIL_NORMAL	список изображений с большими пиктограммами
LVSIL_SMALL	список изображений с малыми пиктограммами
LVSIL_STATE	список с изображениями для отображения состояний

Наличие пиктограмм только одного размера (только малых или только больших) позволяет при помощи масштабирования использовать имеющиеся пиктограммы для всех режимов вывода. В любом случае, для поддержки всех режимов отображения требуется создание обоих списков изображений, причем важно, чтобы они имели одинаковое количество изображений, и соответствующие изображения располагались в одинаковой последовательности, поскольку одно и то же значение (номер по порядку) используется для идентификации необходимой пиктограммы в обоих списках.

Если список состояний изображений задан, то просмотр списка резервирует место слева от каждой пиктограммы в записи для изображения состояния. Приложение может использовать изображения состояний, например, выбрана/не выбрана, для индикации состояния записей, которые определяются приложением. Ненулевое значение в битах с 12 по 15 задает индекс изображения состояния, нуль обозначает отсутствие изображения состояния. В случаях, когда просмотр списка использует режим вывода "большие пиктограммы", изображения состояний обычно не используются.

По умолчанию, когда уничтожается просмотр списка, уничтожаются и его списки изображения. Однако, если просмотр списка имеет стиль LVS_SHAREIMAGELISTS, то приложение само отвечает за уничтожение списка изображений, когда он больше не нужен. Этот стиль, например, следует указывать в том случае, если один список изображений используется для нескольких просмотров списка.

Основные и дополнительные поля

Каждая запись просмотра списка может состоять из изображения, показывающего текущее состояние записи, пиктограммы, основного поля (собственно записи), и значения, определяемого приложением. С каждой записью может быть связано одно или более дополнительных полей. *Дополнительное поле* — это текстовая строка, которая в режиме таблицы (см. выше "Режимы вывода") может быть выведена в столбце справа от пиктограммы и основного поля. Все записи в одном просмотре списка могут иметь одинаковое количество полей. Используя сообщения просмотра списка, можно добавлять, изменять, получать информацию и удалять записи. Можно также находить запись с заданными атрибутами.

Структура *LV_ITEM* определяет записи, включая основное поле, и дополнительные поля просмотра списка. Поле структуры *iItem* задает индекс записи (нумерация начинается с нуля), *iSubItem* задает индекс дополнительного поля (нумерация начинается с единицы). Дополнительные поля задают текст, номер пиктограммы из списка изображений, состояние и данные записи. Под данными записи в данном случае понимается определяемое приложением значение, связанное с записью просмотра списка.

Записи по запросу (Callback Items)

Записи по запросу (иначе, записи обратного вызова) — это записи, для которых приложение (а не элемент управления), сохраняет текст или пиктограммы, или то и другое. Несмотря на то, что просмотр списка может сохранять их, возможно создание записей по запросу, например, для исключения дублирования информации, в частности, текста. Информация для таких записей будет запрашиваться по мере необходимости. Для реализации этого механизма необходимо предусмотреть реализацию обработки уведомлений LVN_GETDISPINFO и LVN_GETDISPINFO.

Кроме того, приложение может взять на себя (временно или постоянно) обработку ряда событий. В этом случае речь идет о задании или изменении маски обратного вызова всего элемента управления. Маска определяет, какие биты состояния записей поддерживаются приложением. По умолчанию маска обратного вызова равна нулю, это означает, что элемент управления самостоятельно отслеживает все изменения состояния записей.

Для получения текущей маски обратного вызова или ее изменения используются две следующие функции:

UINT CListCtrl::**GetCallbackMask**()

и

BOOL CListCtrl::**SetCallbackMask**(UINT nMask)

Параметр *nMask*, а также возвращаемое значение функции *GetCallbackMask* могут содержать любую комбинацию следующих значений:

LVIS_CUT	требует обработки выбора записи списка для выполнения операции вырезания и вставки (cut-and-paste)
LVIS_DROPHILITED	требует обработки назначения целевой записи списка для выполнения операции переноса (drag-and-drop)
LVIS_FOCUSED	требует обработки получения фокуса записи списка
LVIS_SELECTED	требует обработки при выборе записи списка
LVIS_OVERLAYMASK	приложение берет на себя выбор индекса пиктограммы из списка изображений для каждого элемента списка
LVIS_STATEIMAGEMASK	приложение берет на себя выбор индекса пиктограммы из списка изображений состояний для каждого элемента списка

Изменение содержимого просмотра списка

Следующая группа функций предназначена для работы с конкретными записями, а также для их добавления, изменения и удаления.

int CListCtrl::**GetItemCount**() —
возвращает число записей в просмотре списка.

void CListCtrl::**SetItemCount**(int nItems) —
подготавливает просмотр списка к добавлению большого количества новых записей. При этом перед добавлением большого количества записей элемент управления один раз производит перераспределение памяти, вместо того, чтобы делать это всякий раз при добавлении новой записи.

UINT CListCtrl::**GetSelectedCount**() —
возвращает число выбранных записей в просмотре списка (для списков множественного выбора).

BOOL CListCtrl::**DeleteAllItems**() —
удаляет все записи из просмотра списка.

Для того чтобы добавить записи в список, нужно воспользоваться одной из версий функции *InsertItem*. Первая версия использует заранее подготовленную структуру *LV_ITEM*, а последняя в качестве параметров принимает значения, аналогичные значениям полей структуры *LV_ITEM*. Вторая и

третья версии функции полезны для списков, элементы которых не требуют для отображения дополнительных данных:

```
int CListCtrl::InsertItem(const LV_ITEM*pItem)

int CListCtrl::InsertItem(
   int nItem,
   LPCTSTR lpszItem)

int CListCtrl::InsertItem(
   int nItem,
   LPCTSTR lpszItem,
   int nImage)

int CListCtrl::InsertItem(
   UINT nMask,
   int nItem,
   LPCTSTR lpszItem,
   UINT nState,
   UINT nStateMask,
   int nImage,
   LPARAM lParam)
```

Мы подробно рассмотрим только структуру, т. к. имена и типы параметров последней версии функции говорят сами за себя:

```
typedef struct _LV_ITEM {
  UINT mask;
  int iItem;
  int iSubItem;
  UINT state;
  UINT stateMask;
  LPSTR pszText;
  int cchTextMax;
  int iImage;
  LPARAM lParam;
} LV_ITEM;
```

Рассмотрим поля этой структуры.

Поле *mask* задает какие из оставшихся полей несут информацию и может принимать комбинацию следующих значений:

LVIF_TEXT	поле *pszText* содержит данные
LVIF_IMAGE	поле *iImage* содержит данные
LVIF_PARAM	поле *lParam* содержит данные
LVIF_STATE	поле *state* содержит данные

Поле *item* задает индекс записи перед которой осуществляется вставка новой записи; поле *iSubItem* содержит порядковый номер дополнительного поля (в

функции *InsertItem* не используется); поля *state* и *stateMask* задают текущее и возможные состояния для данной записи. Эти поля могут содержать комбинации следующих флагов:

LVIS_CUT	запись выделена для операции вырезания или вставки (cut-and-paste)
LVIS_DROPHILITED	запись выделена для переноса (drag-and-drop)
LVIS_FOCUSED	запись имеет фокус ввода; естественно, что может быть выделено несколько записей одновременно, но только одна может иметь фокус ввода
LVIS_SELECTED	запись выбрана

Поле *pszText* является указателем на текстовую строку (буфер) (если параметр имеет значение LPSTR_TEXTCALLBACK, то это строка обратного вызова); поле *cchTextMax* задает размер буфера, на который указывает поле *pszText*, используется только для получения параметров записи в функции *GetItem* (в функции *InsertItem* значение этого параметра игнорируется); поле *iImage* задает индекс пиктограммы в списке изображений пиктограмм (если параметр имеет значение I_IMAGECALLBACK, то это запись обратного вызова); поле *lParam* задает 32-битное значение, ассоциированное с записью.

Для удаления записи списка по индексу следует использовать функцию *DeleteItem*.

После вставки новой записи, ее атрибуты можно запросить или изменить. Для этого применяются функции, использующие структуру *LV_ITEM*:

BOOL CListCtrl::**GetItem**(LV_ITEM* pItem) —
возвращает атрибуты записи просмотра списка. Для получения текстовых строк следует предусмотреть буфер и указать его размер. В случае если запись содержит строку большего размера, чем размер буфера, она будет усечена.

BOOL CListCtrl::**SetItem**(const LV_ITEM* pItem)

и

```
BOOL CListCtrl::SetItem(
   int nItem,
   int nSubItem,
   UINT nMask,
   LPCTSTR lpszItem,
   int nImage,
   UINT nState,
   UINT nStateMask,
   LPARAM lParam) —
```

устанавливают все или только некоторые атрибуты записи просмотра списка.

Следующие функции позволяют работать с частью атрибутов записи списка без использования структуры *LV_ITEM* и громоздких функций, описанных ранее.

Для получения и назначения 32-битного значения, ассоциированного с записью, предназначены следующие функции:

DWORD CListCtrl::**GetItemData**(int nItem) —

возвращает значение, связанное с записью.

BOOL CListCtrl::**SetItemData**(
int nItem,
DWORD dwData) —

связывает новое значение с записью.

Для получения и программного изменения состояния записи предназначены следующие функции:

UINT CListCtrl::**GetItemState**(
int nItem,
UINT nMask) —

возвращает текущее состояние записи в просмотре списка.

BOOL CListCtrl::**SetItemState**(
int nItem,
LV_ITEM* pItem)

и

BOOL CListCtrl::**SetItemState**(
int nItem,
UINT nState,
UINT nMask) —

изменяют текущее состояние записи в просмотре списка.

Для получения копии текстовой строки записи или, наоборот, установки нового используются следующие функции:

int CListCtrl::**GetItemText**(
int nItem,
int nSubItem,
LPTSTR lpszText,
int nLen)

CString CListCtrl::**GetItemText**(
int nItem,
int nSubItem)

BOOL CListCtrl::**SetItemText**(
int nItem,
int nSubItem,
LPTSTR lpszText) —

позволяют работать не только с основным полем (nSubItem равно 0), но и со всеми дополнительными полями.

Следующие три функции предназначены для управления положением записи в окне просмотра списка:

BOOL CListCtrl::**GetItemRect**(
int nItem,
LPRECT lpRect,
UINT nCode) —
возвращает размеры прямоугольника, ограничивающего запись, где параметр nCode определяет ту часть записи, для которой определяются размеры ограничивающего прямоугольника. Можно использовать следующие флаги:

LVIR_BOUNDS	предписывает возвращать размеры прямоугольника, ограничивающего всю запись, включая пиктограмму и надпись
LVIR_ICON	предписывает возвращать размеры прямоугольника, ограничивающего только пиктограмму (большую или малую)
LVIR_LABEL	предписывает возвращать размеры прямоугольника, ограничивающего текстовую строку (надпись)

BOOL CListCtrl::**GetItemPosition**(
int nItem,
LPPOINT lpPoint) —
возвращает расположение заданной записи в просмотре списка.

BOOL CListCtrl::**SetItemPosition**(
int nItem,
LPPOINT lpPoint) —
помещает запись в заданную позицию в просмотре списка.

Примечание

В разных режимах позиция определяется по-разному, например, в режиме больших пиктограмм начало координат определяется левым верхним углом всего поля, требуемого для вывода всех записей, даже если это поле смещено вверх относительно окна просмотра списка; в режиме таблицы начало координат определяется уже самим окном просмотра списка, и позиция записи может иметь, в том числе, и отрицательные значения по каждой из осей.

Поиск и сортировка записей

Первая функция этой группы предназначена для нахождения записи с определенными свойствами:

int CListCtrl::**GetNextItem**(
int nItem,
int nFlags) —
в случае успешного завершения возвращает индекс найденной записи, иначе -1. Параметр nItem задает индекс записи, с которой начинается поиск, сама она из поиска исключается. Если задано значение -1, то поиск идет с начала списка. Параметр nFlags задает пространственное расположение записи, поиск которой осуществляется, и ее состояние. Расположение может определяться следующими флагами:

LVNI_ABOVE	поиск идет вверх от указанной записи
LVNI_ALL	поиск идет по индексам (по умолчанию)
LVNI_BELOW	поиск идет вниз от указанной записи
LVNI_TOLEFT	поиск идет влево от указанной записи
LVNI_TORIGHT	поиск идет вправо от указанной записи

Состояние может не указываться или быть комбинацией следующих флагов:

LVNI_DROPHILITED	запись имеет установленный флаг LVIS_DROPHILITED
LVNI_FOCUSED	запись имеет установленный флаг LVIS_FOCUSED
LVNI_HIDDEN	запись имеет установленный флаг LVIS_HIDDEN
LVNI_MARKED	запись имеет установленный флаг LVIS_MARKED
LVNI_SELECTED	запись имеет установленный флаг LVIS_SELECTED

Если у записи, которая проверяется на соответствие заданным условиям, установлены не все заданные в параметре *nFlags* флаги, то поиск продолжается со следующей записи.

int CListCtrl::**FindItem**(
  LV_FINDINFO* pFindInfo,
  int nStart = -1) —

осуществляет поиск строки, имеющей заданные характеристики, и возвращает индекс найденной строки или -1. Параметр nStart задает индекс записи, с которой должен начаться поиск, при этом сама она из поиска исключается, или -1, если нужно искать с начала списка. Параметр pFindInfo является указателем на структуру *LV_FINDINFO*, содержащую информацию для осуществления поиска.

Структура *LV_FINDINFO* имеет следующий вид:

```
typedef struct _LV_FINDINFO {
  UINT flags;
  LPCSTR psz;
  LPARAM lParam;
} LV_FINDINFO;
```

Рассмотрим поля этой структуры.

Поле *flags* задает тип поиска. Оно может содержать комбинацию следующих флагов:

LVFI_PARAM	при поиске используется значение, содержащееся в поле *lParam*; если задано данное значение, то все другие значения игнорируются
LVFI_PARTIAL	поиск ведется до совпадения строки, заданной в поле *psz*, и текстового значения записи, т. е. его основного поля; длина текста записи может быть больше, чем строка, заданная в поле *psz*; этот параметр предполагает использование также флага LVFI_STRING

LVFI_STRING	поиск ведется до совпадения строки, заданной в поле *psz*, и текстового значения записи, т. е. его основного поля; если дополнительно не задан флаг LVFI_PARTIAL, то поиск идет до полного совпадения проверяемой строки и строки, указываемой параметром *psz*
LVFI_WRAP	продолжение поиска сначала списка, если строка не найдена по достижению его конца
LVFI_NEARESTXY	задает поиск ближайшей в пространстве поля списка записи от позиции, заданной индексом записи

Поле *psz* является указателем на текстовую строку, которая должна быть найдена, если в параметре *flags* заданы значения LVFI_STRING или LVFI_PARTIAL. Поле *lParam* задает 32-битное значение, которое сравнивается со значениями, ассоциированными с элементами списка.

Просмотр списка в ряде случаев никак не реагирует на действия мышью, если только при этом не происходит изменение состояния его записей. Однако класс *CListCtrl* позволяет определять запись по координатам, и если есть необходимость узнать, например, в обработчике нажатия ее кнопки, на какой записи находится курсор мыши, следует воспользоваться функцией *HitTest*. Однако надо иметь в виду, что, поскольку "система координат" списка зависит от режима отображения,[1] если одновременно не все элементы просмотра списка могут быть отображены на экране, для правильной работы функции *HitTest* следует осуществлять преобразование координат.

int CListCtrl::**HitTest**(LV_HITTESTINFO* pHitTestInfo)

или

int CListCtrl::**HitTest**(
  CPoint pt,
  UINT* pFlags = NULL) —

возвращают индекс записи просмотра списка, расположенной в заданных координатах, если такой записи нет, функция возвращает -1. Обе версии функции отличаются только способом передачи данных для поиска. Рассмотрим первую версию функции, т. к. структура *LV_HITTESTINFO*, указатель на которую передается в параметре pHitTestInfo, содержит поля, аналогичные параметрам второй версии функции HitTest.

Итак, для задания параметров поиска, формируется структура *LV_HITTESTINFO*. Вот ее описание:

```
typedef struct _LV_HITTESTINFO {
  POINT pt;
  UINT flags;
  int iItem;
} LV_HITTESTINFO;
```

[1] См. примечание к описанию функции *SetItemPosition*.

Поле *pt* задает координаты точки, которая предположительно находится в рамках занимаемой некоторым элементом области. Поле *flags* используется для получения дополнительной информации и может быть заполнено комбинацией из следующих флагов:

LVHT_ABOVE	проверяемая точка расположена выше клиентской области элемента управления
LVHT_BELOW	проверяемая точка расположена ниже клиентской области элемента управления
LVHT_NOWHERE	проверяемая точка расположена в клиентской области просмотра списка, но лежит не в области собственно списка
LVHT_ONITEMICON	проверяемая точка расположена в границах пиктограммы записи
LVHT_ONITEMLABEL	проверяемая точка расположена в границах надписи записи
LVHT_ONITEMSTATEICON	проверяемая точка расположена в границах пиктограммы состояния записи
LVHT_TOLEFT	проверяемая точка расположена левее клиентской области элемента управления
LVHT_TORIGHT	проверяемая точка расположена правее клиентской области элемента управления

Флаги LVHT_ABOVE, LVHT_BELOW, LVHT_TOLEFT и LVHT_TORIGHT можно использовать для определения необходимости прокрутки содержимого просмотра списка в соответствующем направлении. Значение LVHT_ONITEM можно использовать для определения того, находится ли вообще точка над записью, это значение является комбинацией значений LVHT_ONITEMICON, LVHT_ONITEMLABEL и LVHT_ONITEMSTATEICON.

Следующая функция осуществляет сортировку записей просмотра списка:

```
BOOL CListCtrl::SortItems(
  PFNLVCOMPARE pfnCompare,
  DWORD dwData)
```

используя определяемую приложением функцию для сравнения записей. Индексы записей при этом изменяются, отражая новый порядок записей в списке. Параметр *pfnCompare* указывает на определяемую приложением функцию сравнения записей. Эта функция вызывается всякий раз в процессе сортировки, когда необходимо сравнить две записи. Второй параметр позволяет передавать в функцию сравнения некоторое значение в качестве третьего параметра. Вот как должна быть описана функция сравнения:

```
int CALLBACK CompareFunc(
  LPARAM lParam1,
  LPARAM lParam2,
  LPARAM lParamSort)
```

первые два параметра содержат 32-битные данные, ассоциированные со сравниваемыми записями, последний параметр является копией значения параметра *dwData* функции *CListCtrl::SortItems*. Функция возвращает отрицательное значение, если первая запись должна предшествовать второй записи, положительное значение, если первая запись должна быть расположена после второй записи, и нуль, если записи эквивалентны.

Работа со столбцами

Если появляется необходимость реализовать поддержку режима отображения содержимого просмотра списка в виде таблицы, следует в обязательном порядке, независимо от количества полей представления записей списка, создать столбцы. Для определения требуемой ширины столбца следует воспользоваться функцией *GetStringWidth*:

```
int CListCtrl::GetStringWidth(LPCTSTR lpsz)
```

которая на основании строки текста позволяет определить минимальную ширину столбца, необходимую для ее вывода на экран целиком. Для добавления (вставки) нового столбца следует использовать функцию *InsertColumn*:

```
int CListCtrl::InsertColumn(
   int nCol,
   const LV_COLUMN* pColumn)
или
int CListCtrl::InsertColumn(
   int nCol,
   LPCTSTR lpszColumnHeading,
   int nFormat = LVCFMT _LEFT,
   int nWidth = -1,
   int nSubItem = -1).
```

Обе версии взаимозависимы. Параметр *nCol* определяет индекс новой колонки. Как обычно, рассмотрим ту реализацию, которая в качестве параметров, в данном случае, для нового столбца, использует структуру. (Однако использование второй версии несколько проще, и в примере, приведенном в конце раздела, используется именно вторая версия функции *InsertColumn*). Эта структура называется *LV_COLUMN* и имеет следующее описание:

```
typedef struct _LV_COLUMN {
   UINT mask;
   int fmt;
   int cx;
   LPSTR pszText;
   int cchTextMax;
   int iSubItem;
} LV_COLUMN;
```

Поле *mask* определяет, какие из остальных полей структуры заданы для описания столбца, и может быть равен нулю или содержать комбинацию следующих флагов:

LVCF_FMT поле *fmt* содержит данные

LVCF_SUBITEM поле *iSubItem* содержит данные

LVCF_TEXT поле *pszText* содержит данные

LVCF_WIDTH поле *cx* содержит данные

Поле *cx* определяет ширину столбца, при значении этого поля, равном −1, ширина не задается, т. е. содержимое столбца не будет отображаться; поле *fmt* задает режим выравнивания столбцов, причем этот режим относится как к заголовку столбца, если он задан и отображается, так и к содержимому полей всех записей просмотра списка (всего таких режимов три, и названия флагов, их определяющих, говорят сами за себя: LVCFMT_LEFT, LVCFMT_RIGHT или LVCFMT_CENTER); поле *pszText* содержит указатель на строку, содержащую наименование заголовка столбца (при получении информации о столбце это поле должно указывать на буфер, куда будет помещено текущее название, длина буфера задается значением поля *cchTextMax*); поле *iSubItem* связывает столбец с полем записи.

Для удаления столбца предназначена функция *DeleteColumn*. Для получения параметров колонки, а также для их изменения предназначены следующие функции.

```
BOOL CListCtrl::GetColumn(
  int nCol,
  LV_COLUMN* pColumn) —
      возвращает параметры столбца.

BOOL CListCtrl::SetColumn(
  int nCol,
  const LV_COLUMN* pColumn) —
      устанавливает новые параметры столбца.

int CListCtrl::GetColumnWidth(int nCol) —
      возвращает ширину столбца.

BOOL CListCtrl::SetColumnWidth(
  int nCol,
  int cx) —
      устанавливает ширину столбца.
```

Редактирование надписей записей

Поддержка пользовательского редактирования надписей записей осуществляется заданием стиля LVS_EDITLABELS и обработкой уведомления LVN_ENDLABELEDIT; что и отражено в сопровождающем раздел примере.

Обработчик уведомления получает указатель на новую строку, которую ввел пользователь. Для того чтобы фактически осуществить замену предыдущего наименования, следует самостоятельно заменить содержимое записи, а также при необходимости произвести дополнительные действия, например, при изменении копии имени файла, содержащейся в записи просмотра списка, требуется также изменить его имя и на диске или ином носителе. В случае, если значение указателя равно нулю, — пользователь отменил изменения, нажав клавишу <Esc> — ничего делать не придется.

Класс *CListCtrl* предоставляет дополнительные функции, расширяющие возможности поддержки процесса редактирования надписей записей.

CEdit* CListCtrl::**GetEditControl**() —
возвращает дескриптор редактора текста, используемого для редактирования надписи записи. В случае удачного завершения возвращается указатель на объект класса CEdit, в случае неудачи — NULL.

CEdit* CListCtrl::**EditLabel**(int nItem) —
позволяет программно инициировать редактирование надписи записи в просмотре списка, если просмотр списка имеет стиль LVS_EDITLABELS.

Обработка уведомлений

Когда пользователь щелкает на колонке заголовков, перемещает пиктограммы, редактирует надписи и т. п. — просмотр списка посылает родительскому окну уведомления. Их возможно обрабатывать в функции родительского окна *OnNotify*.

Просмотр списка посылает следующие уведомления:

LVN_BEGINDRAG	сигнализирует о начале операции перемещения при помощи левой кнопки мыши (drag-and-drop)
LVN_BEGINLABELEDIT	сигнализирует о начале операции редактирования надписи записи
LVN_BEGINRDRAG	сигнализирует о начале операции перемещения при помощи правой кнопки мыши (drag-and-drop)
LVN_COLUMNCLICK	сигнализирует о том, что пользователь нажал кнопку мыши, когда курсор находился на элементе заголовка в режиме таблицы
LVN_DELETEALLITEMS	сигнализирует об удалении всех записей просмотра списка
LVN_DELETEITEM	сигнализирует об удалении конкретной записи просмотра списка
LVN_ENDLABELEDIT	сигнализирует о завершении операции редактирования надписи записи
LVN_GETDISPINFO	запрашивает информацию о записи по запросу просмотра списка для ее отображения

LVN_INSERTITEM	сигнализирует о добавлении новой записи к просмотру списка
LVN_ITEMCHANGED	сигнализирует об изменении записи просмотра списка
LVN_ITEMCHANGING	запрашивает подтверждение изменения записи просмотра списка
LVN_KEYDOWN	сигнализирует о нажатии клавиши на клавиатуре
LVN_SETDISPINFO	уведомляет родительское окно о необходимости изменения информации о записи по запросу, которая предоставляется приложением

Обработка уведомлений происходит как и для других элементов управления, за исключением того, что приложение получает информацию в структуре *LV_DISPINFO*, которая, кроме обычного для уведомлений заголовка, содержит структуру *LV_ITEM* со всей необходимой для обработки информацией:

```
BOOL CMercuryWnd::OnNotify(
  WPARAM wParam, LPARAM lParam, LRESULT* pResult)
{
  ...
  LV_DISPINFO *pdi = (LV_DISPINFO *)lParam;
  LV_ITEM item = pdi->item;
  ...
}
```

Некоторые уведомляющие сообщения работают с другой структурой, а именно со структурой *NM_LISTVIEW*:

```
NM_LISTVIEW *plv = (NM_LISTVIEW *)lParam;
```

Реализация просмотра списка с возможностью перемещения записей

Перемещение записей возможно реализовать только для режимов больших и малых пиктограмм, что вполне естественно. Главное, что нужно для этого помнить, это то, что индексы записей после изменения положения записей не будут отражать их фактический порядок на экране — порядок размещения записей в просмотре списка автоматически не меняется.

Для поддержки перемещения записей просмотра списка требуется обрабатывать следующие уведомления: LVN_BEGINDRAG, LVN_BEGINRDRAG, а также сообщения от мыши: отпускание левой и правой кнопок, а также перемещение курсора.

Кроме того, следует создать еще один список изображений, который будет содержать элемент для визуализации перемещения записи списка при помощи функции *CreateDragImage*:

CImagelist* CListCtrl::**CreateDragImage**(
int nItem,
LPPOINT lpPoint)

параметр nItem задает при помощи индекса запись, а параметр lpPoint задает текущие координаты левого верхнего угла записи.

Переопределяемые функции

Для реализации самоотображаемого списка следует переопределить функцию *DrawItem*, которая, как и для других элементов управления, получает всю необходимую информацию для осуществления рисования записей просмотра списка.

ListCtrl — пример приложения

В этом разделе приведен пример использования просмотра списка, в качестве записей которого используется информация о файлах текущего (для приложения) каталога, а именно — имя, тип (определяемый по расширению), длина в килобайтах, а также дата и время последней операции записи.

Приложение реализует практически все рассмотренные в предыдущих разделах механизмы, включая:

- возможность отображения во всех четырех режимах просмотра;
- сортировку по всем четырем используемым атрибутам файла;
- редактирование в режиме таблицы надписи записи (файла) без записи изменений на диск;
- перемещение записей при помощи мыши;
- переупорядочивание записей, необходимое после выполнения операции перемещения.

```
/////////////////////////////////////////////////////////////
Файл ListCtrlFrm.cpp
Copyright (c) 1997 Мешков А., Тихомиров Ю., Шведов Д.
/////////////////////////////////////////////////////////////

// ListCtrlFrm.cpp : implementation file
//

#include "stdafx.h"
#include "ListCtrl.h"
#include "ListCtrlFrm.h"

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

```
// функция сортировки записей,
// тип сортировки определяется
// значением третьего параметра
int CALLBACK SortFunc(LPARAM lParam1,
  LPARAM lParam2, LPARAM dwType)
{
  WIN32_FIND_DATA *item1 = (WIN32_FIND_DATA *)lParam1;
  WIN32_FIND_DATA *item2 = (WIN32_FIND_DATA *)lParam2;

  switch (dwType)
  {
  case ID_VIEW_ARRANGE_BYNAME:
    return strcmp(item1->cFileName, item2->cFileName);
  case ID_VIEW_ARRANGE_BYTYPE:
    return strcmp(strrchr(item1->cAlternateFileName, '.'),
      strrchr(item2->cAlternateFileName, '.'));
  case ID_VIEW_ARRANGE_BYSIZE:
    return item1->nFileSizeLow - item2->nFileSizeLow;
  case ID_VIEW_ARRANGE_BYDATE:
    return CompareFileTime(&item1->ftLastWriteTime,
      &item2->ftLastWriteTime);
  default:
    ASSERT(FALSE);
  }

  return 0;
}

////////////////////////////////////////////////////////////
// CListCtrlFrame

CListCtrlFrame::CListCtrlFrame()
{
  // инициализация переменных, используемых
  // для реализации перемещения записей
  m_bDragging = FALSE;
  m_pImageListDrag = NULL;
}

CListCtrlFrame::~CListCtrlFrame()
{
}

// функция изменения стиля просмотра списка
// для смены режимов отображения
void CListCtrlFrame::SetStyle(DWORD dwViewStyle)
{
  ASSERT(::IsWindow(m_List.m_hWnd));

  DWORD dwStyle = GetWindowLong(m_List.m_hWnd, GWL_STYLE);
```

```
  if ((dwStyle & LVS_TYPEMASK) != dwViewStyle)
    SetWindowLong(m_List.m_hWnd, GWL_STYLE,
    (dwStyle & ~LVS_TYPEMASK) | dwViewStyle);
}

BEGIN_MESSAGE_MAP(CListCtrlFrame, CFrameWnd)
  //{{AFX_MSG_MAP(CListCtrlFrame)
  ON_WM_CREATE()
  ON_COMMAND(ID_VIEW_LARGEICONS, OnViewLargeIcons)
  ON_COMMAND(ID_VIEW_SMALLICONS, OnViewSmallIcons)
  ON_COMMAND(ID_VIEW_LIST, OnViewList)
  ON_COMMAND(ID_VIEW_DETAILS, OnViewDetails)
  ON_COMMAND(ID_VIEW_ARRANGE_BYNAME, OnViewArrangeByName)
  ON_COMMAND(ID_VIEW_ARRANGE_BYTYPE, OnViewArrangeByType)
  ON_COMMAND(ID_VIEW_ARRANGE_BYSIZE, OnViewArrangeBySize)
  ON_COMMAND(ID_VIEW_ARRANGE_BYDATE, OnViewArrangeByDate)
  ON_COMMAND(ID_VIEW_ARRANGE_AUTOARRANGE,
    OnViewArrangeAutoArrange)
  ON_WM_MOUSEMOVE()
  ON_WM_LBUTTONUP()
  ON_WM_RBUTTONUP()
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////
// CListCtrlFrame message handlers

int CListCtrlFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
  if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
    return -1;

  CRect rect;
  GetClientRect(rect);

  // создание и инициализация просмотра списка
  m_List.Create(LVS_ICON | LVS_REPORT | LVS_EDITLABELS |
    WS_VISIBLE, rect, this, 1000);

  // создание и заполнение списков изображений
  m_ImageNormal.Create(32, 32, TRUE, 1, 1);
  m_ImageNormal.Add(AfxGetApp()->LoadIcon(IDI_FILE));

  m_ImageSmall.Create(16, 16, TRUE, 1, 1);
  m_ImageSmall.Add(AfxGetApp()->LoadIcon(IDI_FILE));

  m_List.SetImageList(&m_ImageNormal, LVSIL_NORMAL);
  m_List.SetImageList(&m_ImageSmall, LVSIL_SMALL);

  // добавление в просмотр списка четырех колонок для
  // реализации режима таблицы
  m_List.InsertColumn(0, _T("Name"),
    LVCFMT_LEFT, rect.Width() / 3);
```

```
m_List.InsertColumn(1, _T("Type"),
  LVCFMT_LEFT, 2 * rect.Width() / 9);
m_List.InsertColumn(2, _T("Size"),
  LVCFMT_RIGHT, rect.Width() / 9);
m_List.InsertColumn(3, _T("Modified"),
  LVCFMT_LEFT, rect.Width() / 3);

// поиск файлов и создание записей с
// информацией о найденных файлах
WIN32_FIND_DATA FindFileData;

HANDLE hFile = FindFirstFile("*.*", &FindFileData);

if (hFile != INVALID_HANDLE_VALUE)
{
  LV_ITEM item;
  item.iItem = 0;
  item.iImage = 0;

  CString csText;
  SYSTEMTIME stTime;

  do
  {
    // каталоги игнорируем
    if (FindFileData.dwFileAttributes &
      FILE_ATTRIBUTE_DIRECTORY)
      continue;

    item.mask = LVIF_TEXT | LVIF_IMAGE | LVIF_PARAM;

    // для реализации сортировки дополнительно сохраняем
    // информацию о найденном файле
    WIN32_FIND_DATA *lpFindFileData = new WIN32_FIND_DATA;
    *lpFindFileData = FindFileData;

    item.lParam = (DWORD)lpFindFileData;

    // имя файла
    item.iSubItem = 0;
    item.pszText = FindFileData.cFileName;
    m_List.InsertItem(&item);

    item.mask = LVIF_TEXT;

    // тип файла
    item.iSubItem = 1;
    csText = (strrchr(FindFileData.cAlternateFileName, '.')
      + 1);
    csText += " File";
    item.pszText = (LPTSTR)(LPCTSTR)csText;
    m_List.SetItem(&item);
```

```
      // длина файла
      item.iSubItem = 2;
      csText.Format("%dKB", FindFileData.nFileSizeLow / 1024);
      item.pszText = (LPTSTR)(LPCTSTR)csText;
      m_List.SetItem(&item);

      // дата и время последней записи в файл
      item.iSubItem = 3;
      FileTimeToSystemTime(&FindFileData.ftLastWriteTime,
        &stTime);
      csText = CTime(stTime).Format(" %d/%m/%Y %H:%M:%S");
      item.pszText = (LPTSTR)(LPCTSTR)csText;
      m_List.SetItem(&item);

      item.iItem++;
    }
    while (FindNextFile(hFile, &FindFileData));
  }

  return 0;
}
```

После загрузки приложения появится следующее окно:

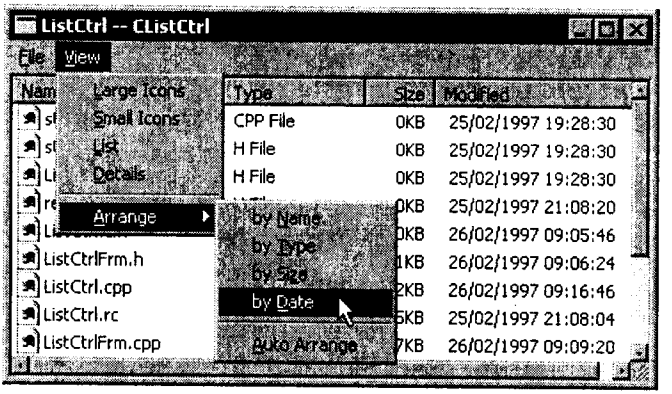


Рис. 29. Начальное состояние окна приложения ListCtrl с демонстрацией реализованных возможностей

```
//
// функции смены режима отображения
//

void CListCtrlFrame::OnViewLargeIcons()
{
  SetStyle(LVS_ICON);
}

void CListCtrlFrame::OnViewSmallIcons()
```

```
{
  SetStyle(LVS_SMALLICON);
}
void CListCtrlFrame::OnViewList()
{
  SetStyle(LVS_LIST);
}
void CListCtrlFrame::OnViewDetails()
{
  SetStyle(LVS_ICON | LVS_REPORT);
}
//
// функции переупорядочивания (сортировки) по значениям
// различным атрибутов файла
//
void CListCtrlFrame::OnViewArrangeByName()
{
  m_List.SortItems(SortFunc, ID_VIEW_ARRANGE_BYNAME);
}
void CListCtrlFrame::OnViewArrangeByType()
{
  m_List.SortItems(SortFunc, ID_VIEW_ARRANGE_BYTYPE);
}
void CListCtrlFrame::OnViewArrangeBySize()
{
  m_List.SortItems(SortFunc, ID_VIEW_ARRANGE_BYSIZE);
}
void CListCtrlFrame::OnViewArrangeByDate()
{
  m_List.SortItems(SortFunc, ID_VIEW_ARRANGE_BYDATE);
}
// функция переупорядочивания - привязки записей списка
// к сетке привязки
void CListCtrlFrame::OnViewArrangeAutoArrange()
{
  m_List.Arrange(LVA_SNAPTOGRID);
}
//
// функция обработки извещений для поддержки:
//   - редактирования надписей записей;
//   - перемещения записей
//
BOOL CListCtrlFrame::OnNotify(
  WPARAM wParam, LPARAM lParam, LRESULT* pResult)
```

```
{
  NMHDR *nmhdr = (NMHDR *)lParam;
  if (nmhdr->hwndFrom == m_List.m_hWnd)
  {
    LV_DISPINFO *pdi = (LV_DISPINFO *)lParam;
    // для поддержки перемещения записей,
    // необходимо работать
    // со структурой NM_LISTVIEW
    NM_LISTVIEW *plv = (NM_LISTVIEW *)lParam;
    CPoint ptItem, ptAction, ptImage;
    switch (nmhdr->code)
    {
    case LVN_ENDLABELEDIT:
      if (pdi->item.pszText != NULL)
        m_List.SetItemText(pdi->item.iItem, 0, pdi->item.pszText);
      break;
    case LVN_BEGINDRAG:
    case LVN_BEGINRDRAG:
      ASSERT(!m_bDragging);
      m_bDragging = TRUE;
      m_iItemDrag = plv->iItem;
      ptAction = plv->ptAction;
      m_List.GetItemPosition(m_iItemDrag, &ptItem);
      m_List.GetOrigin(&m_ptOrigin);
      m_pImageListDrag =
        m_List.CreateDragImage(m_iItemDrag, &ptImage);
      m_sizeDelta = ptAction - ptImage;
      m_ptHotSpot = ptAction - ptItem + m_ptOrigin;
      m_pImageListDrag->DragShowNolock(TRUE);
      m_pImageListDrag->SetDragCursorImage(0, m_ptHotSpot);
      m_pImageListDrag->BeginDrag(0, CPoint(0, 0));
      ptAction -= m_sizeDelta;
      m_pImageListDrag->DragEnter(&m_List, ptAction);
      m_pImageListDrag->DragMove(ptAction);
      SetCapture();
      break;
    }
  }
  return CFrameWnd::OnNotify(wParam, lParam, pResult);
}
//
// функции обработки сообщений от мыши
//
void CListCtrlFrame::OnMouseMove(UINT nFlags, CPoint point)
{
  if (m_bDragging)
```

```
    m_pImageListDrag->DragMove(point - m_sizeDelta);
  CFrameWnd::OnMouseMove(nFlags, point);
}
void CListCtrlFrame::OnButtonUp(CPoint point)
{
  if (m_bDragging)
  {
    DWORD  dwStyle;
    dwStyle = GetWindowLong(m_List.m_hWnd, GWL_STYLE) &
      LVS_TYPEMASK;
    m_bDragging = FALSE;
    m_pImageListDrag->DragLeave(&m_List);
    m_pImageListDrag->EndDrag();
    if (dwStyle & (LVS_REPORT | LVS_LIST))
      MessageBeep((UINT)-1);
    else if (dwStyle == LVS_ICON || dwStyle == LVS_SMALLICON)
    {
      point -= m_ptHotSpot;
      point += m_ptOrigin;
      m_List.SetItemPosition(m_iItemDrag, point);
    }
    ReleaseCapture();
  }
}
void CListCtrlFrame::OnLButtonUp(UINT nFlags, CPoint point)
{
  OnButtonUp(point);
  CFrameWnd::OnLButtonUp(nFlags, point);
}
void CListCtrlFrame::OnRButtonUp(UINT nFlags, CPoint point)
{
  OnButtonUp(point);
  CFrameWnd::OnRButtonUp(nFlags, point);
}
// т. к. было осуществлено динамическое выделение памяти
// для реализации хранения информации о файлах,
// при уничтожении просмотра списка
// необходимо освободить эту память
BOOL CListCtrlFrame::DestroyWindow()
{
  for (int i = 0; i < m_List.GetItemCount(); i++)
    delete (WIN32_FIND_DATA *)m_List.GetItemData(i);
  return CFrameWnd::DestroyWindow();
}
```

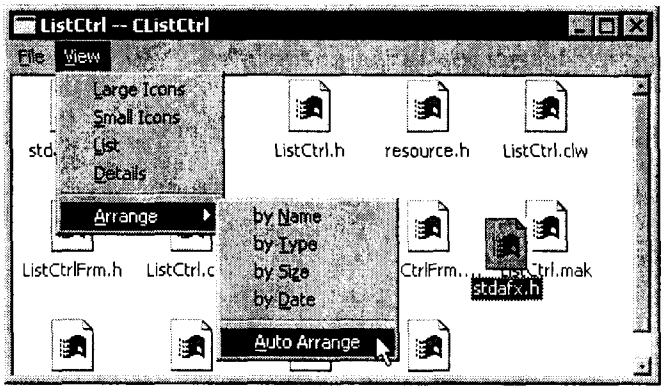
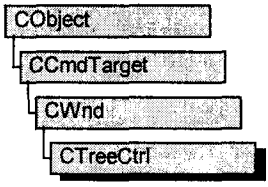


Рис. 30. Окно приложения ListCtrl в режиме больших пиктограмм после осуществления перемещения записи и перед переупорядочиванием

Просмотр дерева (класс CTreeCtrl)



Кроме возможности выводить записи последовательных списков при помощи таких элементов, как список или просмотр списка, Windows предоставляет возможность отображения (и управления) списков, организованных по принципу дерева. Для этого поддерживается элемент управления "просмотр дерева" (TREE VIEW). Первое, для чего этот элемент управления использует система Windows — это организация просмотра дерева каталогов, а также различных индексов в справочных файлах и т. п.

Древовидные структуры предполагают наличие некоторой иерархии узлов и, естественно, что данный элемент управления может применяться только для информации, имеющей иерархическую структуру. Для создания этого элемента управления в библиотеке MFC предназначен класс *CTreeCtrl*. Каждая запись просмотра дерева состоит из текстовой строки (надписи) и заданного битового изображения, каждая запись может иметь одну или более подзаписей, связанных с ней. Щелкнув мышью по записи, пользователь может развернуть или свернуть список связанных с записью подзаписей. На экране подзаписи могут соединяться с записью линиями для того, чтобы наглядно представлялась их взаимосвязь.

Любая запись просмотра дерева может иметь связанные с ней подзаписи, которые называются *дочерними записями*, а сама запись называется *родительской записью*. При выводе на экран дочерние записи изображаются ниже родительской и со сдвигом в сторону для наглядного представления их зависимости от родительской записи. Запись, которая не имеет родительской и, следовательно, лежит на самом верхнем уровне иерархии, называется *корневой записью*.

Кроме класса *CTreeCtrl*, библиотека MFC содержит еще и класс *CTreeView*. Класс *CTreeCtrl* предназначен для создания просмотра дерева в блоке диалога, в особенности при наличии в блоке диалога других элементов управления. Напротив, если просмотр дерева должен быть создан в архитектуре документ/представление и будет занимать собой все окно, автоматически изменяя свои размеры при изменении размеров родительского окна, обрабатывать сообщения от меню и т. п., то необходимо использовать класс *CTreeView*.

Создание просмотра дерева

Для начала просмотр дерева необходимо создать. Если он задан в шаблоне блока диалога, то он будет автоматически создан при создании блока диалога. Если же он создается как дочернее окно в другом окне, то необходимо воспользоваться функцией *Create* для создания соответствующего элемента управления Windows и связывания его с объектом класса *CTreeCtrl*:

```
BOOL CTreeCtrl::Create(
   DWORD dwStyle,
   const RECT& rect,
   CWnd* pParentWnd,
   UINT nID)
```

Параметр *dwStyle* может содержать любые комбинации стилей элементов управления, а также следующие, свойственные для рассматриваемого элемента управления:

TVS_HASLINES	дочерние записи будут соединяться на экране линиями со своими родительскими записями
TVS_LINESATROOT	записи первого уровня будут соединены с вершиной дерева, в противном случае, они всегда будут отображать свои дочерние записи

Применение обоих из этих стилей придает содержимому элемента управления вид древовидной структуры.

TVS_HASBUTTONS	добавляет слева от каждой записи кнопку для сворачивания-разворачивания ветки дочерних записей (если для нее допустима операция разворачивания; кнопка содержит знак "+", если запись можно развернуть до следующего уровня, и знак "-", если ветвь дерева развернута, при этом дочерние записи могут находиться в свернутом состоянии
TVS_EDITLABELS	позволяет пользователю редактировать записи непосредственно в просмотре дерева
TVS_SHOWSELALWAYS	выделение с выбранной записи не снимается, даже если просмотр дерева теряет фокус ввода
TVS_DISABLEDRAGDROP	просмотр дерева препятствует посыланию уведомления TVN_BEGINDRAG

Параметр *rect* задает размер и расположение просмотра дерева. Параметр *pParentWnd* задает родительское окно. Значение этого параметра не должно быть равно NULL. Параметр *nID* задает идентификатор просмотра дерева.

Списки изображений

Каждая запись просмотра дерева может иметь две ассоциированных с ней пиктограммы. Первая пиктограмма выводится на экран, когда запись развернута, вторая — когда запись свернута (например, пиктограммы, изображающие открытый и закрытый каталог).

Кроме двух списков, описанных выше, приложение может задать список изображений для обозначения состояний записи. В этом случае просмотр дерева резервирует место слева от записи для размещения данного изображения.

Для работы с изображениями, связанными с просмотром дерева, предназначены функции:

CImageList* CTreeCtrl::**GetImageList**(UINT nImage) —
возвращает указатель на список изображений, связанных с просмотром дерева, или NULL в случае неудачного завершения. Параметр nImage задает тип списка и может принимать одно из следующих значений:

TVSIL_NORMAL	предписывает возвращать список обычных изображений, который содержит изображения для выбранных и невыбранных записей
TVSIL_STATE	предписывает возвращать список изображений, который содержит изображения для определяемых пользователем состояний

CImageList* CTreeCtrl::**SetImageList**(
CImageList* pImageList,
int nImageListType) —
устанавливает новый список изображений для просмотра дерева, тип списка задается параметром nImageListType.

Функции для работы с просмотром дерева в целом

Когда ветка дочерних записей развернута, дочерние записи изображаются с некоторым сдвигом относительно родительской записи. Для определения и изменения сдвига дочерней записи относительно родительской предназначены функции:

UINT CTreeCtrl::**GetIndent**() —
возвращает сдвиг (в пикселях) дочерней записи относительно родительской.

void CTreeCtrl::**SetIndent**(UINT nIndent) —
задает сдвиг (в пикселях) дочерней записи относительно родительской. Если задается значение, меньшее чем определяемое системой, то функция игнорируется.

Изменения содержимого дерева

Следующая группа функций предназначена для работы с конкретными записями, а также для их добавления, изменения и удаления.

UINT CTreeCtrl::**GetCount**() —
возвращает число записей, вставленных в просмотр дерева.

UINT CTreeCtrl::**GetVisibleCount**() —
возвращает текущее количество видимых записей просмотра дерева.

Для того чтобы удалить все записи из просмотра дерева, можно воспользоваться функцией *DeleteAllItems*.

Начальное положение записи при ее добавлении в просмотр дерева определяется с помощью функции *InsertItem*. При вызове этой функции задаются дескриптор родительской записи и дескриптор записи, после которой должна быть вставлена новая запись. Второй из указанных дескрипторов должен быть дескриптором одной из дочерних записей, принадлежащих родительской записи, или принимать одно из следующих значений: TVI_FIRST, TVI_LAST или TVI_SORT. Значения TVI_FIRST и TVI_LAST задают, что запись должна быть помещена первой или последней в списке дочерних записей родительской записи. Если задано значение TVI_SORT, то после вставки записи, дочерние записи принадлежащие данной родительской записи, будут отсортированы по алфавиту.

Функция, осуществляющая вставку новых записей, реализована в виде нескольких версий:

```
TREEITEM CTreeCtrl::InsertItem(LPTV_INSERTSTRUCTlpInsertStruct)

HTREEITEM CTreeCtrl::InsertItem(
  UINT nMask,
  LPCTSTR lpszItem,
  int nImage,
  int nSelectedImage,
  UINT nState,
  UINT nStateMask,
  LPARAM lParam,
  HTREEITEM hParent,
  HTREEITEM hInsertAfter)

HTREEITEM CTreeCtrl::InsertItem(
  LPCTSTR lpszItem,
  HTREEITEM hParent = TVI_ROOT,
  HTREEITEM hInsertAfter = TVI_LAST)

HTREEITEM CTreeCtrl::InsertItem(
  LPCTSTR lpszItem,
  int nImage,
  int nSelectedImage,
  HTREEITEM hParent = TVI_ROOT,
  HTREEITEM hInsertAfter = TVI_LAST)
```

Все версии этой функции вставляют новую запись в просмотр дерева и возвращают дескриптор вставленной записи. Первые два варианта практически аналогичны по своим возможностям. Две оставшиеся обеспечивают просмотр дерева лишь частью информации о новой записи. Однако их целесообразно использовать в относительно простых реализациях, когда от просмотра дерева не требуется поддержка всех функциональных возможностей. Кроме того, существуют функции, позволяющие впоследствии дополнить информацию о записи.

Чтобы не дублировать описание, приведем более подробно только описание структур *TV_INSERTSTRUCT* и *TV_ITEM*, которые вместе содержат полный набор параметров, описывающих добавляемую запись, и необходимых для реализации добавления.

Структура *TV_INSERTSTRUCT* имеет следующий вид:

```
typedef struct _TV_INSERTSTRUCT {
  HTREEITEM hParent;
  HTREEITEM hInsertAfter;
  TV_ITEM item;
} TV_INSERTSTRUCT;
```

Поле *hParent* задает дескриптор родительской записи. Если параметр равен TVI_ROOT или NULL, то запись вставляется как корневая. Поле *hInsertAfter* задает дескриптор записи, после которой запись должна быть вставлена. Оно может принимать также следующие значения:

TVI_FIRST запись вставляется в начало списка

TVI_LAST запись вставляется в конец списка

TVI_SORT запись вставляется в список в алфавитном порядке

Поле *item* является структурой *TV_ITEM*, именно в этом поле содержится информация о добавляемой записи. Структура *TV_ITEM* имеет следующий вид:

```
typedef struct _TV_ITEM {
  UINT mask;
  HTREEITEM hItem;
  UINT state;
  UINT stateMask;
  LPSTR pszText;
  int cchTextMax;
  int iImage;
  int iSelectedImage;
  int cChildren;
  LPARAM lParam;
} TV_ITEM;
```

Рассмотрим поля этой структуры.

Поле *mask* определяет, какие из оставшихся полей несут информацию, и может содержать комбинацию следующих значений:

TVIF_CHILDREN	поле *cChildren* содержит данные
TVIF_HANDLE	поле *hItem* содержит данные
TVIF_IMAGE	поле *iImage* содержит данные
TVIF_PARAM	поле *lParam* содержит данные
TVIF_SELECTEDIMAGE	поле *iSelectedImage* содержит данные
TVIF_STATE	оба поля *state* и *stateMask* содержат данные
TVIF_TEXT	поле *pszText* содержит данные

Поле *hItem* содержит дескриптор записи, информация о которой содержится в структуре. Разумеется, при добавлении новой записи этот дескриптор не задается, т. е. поле используется для запроса информации о существующей записи.

Поля *state* и *stateMask* задают текущее и возможные состояния для данной записи. Эти поля могут содержать комбинации следующих флагов (за исключением двух последних):

TVIS_CUT	запись помечена для операции вырезания или вставки (cut-and-paste)
TVIS_DISABLED	запись недоступна и для ее перерисовки применяется стандартный стиль и цвет
TVIS_DROPHILITED	запись выделена для переноса (drag-and-drop)
TVIS_EXPANDED	список дочерних записей должен быть развернут так, чтобы дочерние записи были видимы; данное значение применимо только для родительской записи
TVIS_EXPANDEDONCE	список дочерних записей был развернут по крайней мере один раз; для родительской записи, имеющей данный стиль, уведомления TVN_ITEMEXPANDING и TVN_ITEMEXPANDED не посылаются; данное значение применимо только для родительской записи
TVIS_FOCUSED	запись имеет фокус ввода; естественно, что может быть выделено несколько записей одновременно, но фокус ввода может иметь только одна
TVIS_SELECTED	запись выбрана
TVIS_OVERLAYMASK	подгружаемое изображение для записи включается, когда запись перерисовывается; индекс подгружаемого изображения должен быть задан в параметре *state* структуры *TV_ITEM* при использовании макроса INDEXTOOVERLAYMASK; подгружаемая запись должна быть добавлена в список изображений дерева с помощью функции *CImageList::SetOverlayImage*; это значение не может применяться совместно с другими значениями

TVIS_STATEIMAGEMASK изображение состояния записи включается, когда запись перерисовывается; индекс состояния записи должен быть задан в параметре *state* структуры *TV_ITEM* при использовании макроса INDEXTOSTATEIMAGEMASK; это значение не может применяться совместно с другими значениями

Поле *pszText* является указателем на строку, содержащую надпись записи, если заданы соответствующие атрибуты в структуре. Если параметр имеет значение LPSTR_TEXTCALLBACK, то это строка обратного вызова[1] и родительское окно само отвечает за хранение текстовой строки. В этом случае просмотр дерева посылает родительскому окну уведомление TVN_GETDISPINFO, когда необходимо вывести на экран, отсортировать или отредактировать надпись, и посылает уведомление TVN_SETDISPINFO, если надпись записи изменена. Если структура должна получать атрибуты записи, то это указатель на буфер, в который она получит надпись.

Поле *cchTextMax* определяет размер буфера, на который указывает параметр *pszText*, если структура получает атрибуты записи. Если же она задает атрибуты записи, то данный параметр игнорируется.

Поля *iImage* и *iSelectedImage* задают индексы пиктограмм для невыбранного и выбранного состояний. Если значения одного или обоих полей равны I_IMAGECALLBACK, то родительское окно отвечает за хранение изображений.

Поле *cChildren* содержит количество дочерних записей, связанных с данной записью. Если значение данного поля равно I_CHILDRENCALLBACK, то родительское окно отвечает за перерисовку дочерних записей. Это поле заполняется функцией, возвращающей информацию о записи.

Поле *lParam* задает некоторое 32-битное значение, ассоциируемое с записью.

Для удаления ранее вставленной записи используется функция *DeleteItem*:

BOOL CTreeCtrl::**DeleteItem**(HTREEITEM hItem) —
обратите внимания, что вместо индексов записей используются их дескрипторы. Если параметр hItem равен TVI_ROOT, то будут удалены все записи.

Для изменения или получения всех или части параметров записи предназначены следующие функции:

```
BOOL CTreeCtrl::SetItem(TV_ITEM* pItem)
и
BOOL CTreeCtrl::SetItem(
  HTREEITEM hItem,
  UINT nMask,
  LPCTSTR lpszItem,
```

[1] Подробнее о записях обратного вызова см. "Записи по запросу" в разделе, посвященном просмотрам списка.

int nImage,
int nSelectedImage,
UINT nState,
UINT nStateMask,
LPARAM lParam) —
устанавливают параметры записи в просмотре дерева. Назначение и тип параметров аналогичны рассмотренным ранее.

BOOL CTreeCtrl::**GetItem**(TV_ITEM* pItem) —
возвращает параметры записи в структуре, на которую указывает параметр pItem.

Каждая запись просмотра дерева имеет текущее состояние, например, она может быть выбрана, недоступна, развернута и т. д. Во многих случаях просмотр дерева автоматически устанавливает состояния записей, отражая действия, производимые пользователем. Для того чтобы узнать или изменить только состояние записи, можно воспользоваться следующими функциями:

UINT CTreeCtrl::**GetItemState**(
HTREEITEM hItem,
UINT nStateMask) —
возвращает состояние записи.

BOOL CTreeCtrl::**SetItemState**(
HTREEITEM hItem,
UINT nState,
UINT nStateMask) —
устанавливает состояние записи.

Для того чтобы узнать или изменить только изображения, связанные с записью, можно воспользоваться следующими функциями:

BOOL CTreeCtrl::**GetItemImage**(
HTREEITEM hItem,
int& nImage,
int& nSelectedImage) —
записывает номера изображений в параметры nImage и nSelectedImage.

BOOL CTreeCtrl::**SetItemImage**(
HTREEITEM hItem,
int nImage,
int nSelectedImage) —
устанавливает изображения для заданной записи.

Для того чтобы узнать или изменить только надпись записи, можно воспользоваться следующими функциями:

CString CTreeCtrl::**GetItemText**(HTREEITEM hItem) —
возвращает объект типа CString, содержащий надпись записи.

BOOL CTreeCtrl::**SetItemText**(
HTREEITEM hItem,
LPCTSTR lpszItem) —
заменяет текущее название записи новым.

Следующая пара функций предназначена для получения и изменения 32-битного значения, ассоциированного с записью просмотра дерева:

DWORD CTreeCtrl::**GetItemData**(HTREEITEM hItem) —
позволяет получить 32-битное значение.

BOOL CTreeCtrl::**SetItemData**(
HTREEITEM hItem,
DWORD dwData) —
устанавливает 32-битное значение, заданное параметром dwData.

В случае, когда необходимо получить размеры прямоугольника, ограничивающего запись просмотра дерева, и узнать при этом, видима запись или нет, используется функция *GetItemRect*:

BOOL CTreeCtrl::**GetItemRect**(
HTREEITEM hItem,
LPRECT lpRect,
BOOL bTextOnly) —
возвращает TRUE, если запись hItem видима, в этом случае размеры прямоугольника содержатся в переменной lpRect. Координаты задаются относительно левого верхнего угла окна просмотра дерева. Если значение параметра bTextOnly равно TRUE, то прямоугольник содержит только размеры надписи для данной записи. Иначе, он содержит размеры прямоугольника, который содержит запись целиком.

Следующая описываемая функция возвращает значение TRUE, если указанная запись имеет дочерние записи:

BOOL CTreeCtrl::**ItemHasChildren**(HTREEITEM hItem)

Функцией *GetChildItem* стоит воспользоваться, если нужно получить дескриптор первой дочерней записи:

HTREEITEM CTreeCtrl::**GetChildItem**(HTREEITEM hItem)

Записи могут быть свернуты или развернуты простым щелчком мыши. Если дочерние записи некоторой записи развернуты, то они выводятся ниже ее. Если они свернуты, то они на экран не выводятся. Переключение из одного состояния в другое происходит автоматически при двойном щелчке мыши на записи или нажатием на специальную кнопку, расположенную слева от родительской записи (если запись имеет стиль TVS_HASBUTTONS). Приложение также может разворачивать и сворачивать ветви дерева путем вызова функции *Expand*:

BOOL CTreeCtrl::**Expand**(
HTREEITEM hItem,
UINT nCode) —

сворачивает или разворачивает дочерние записи, той, что задана параметром hItem, параметр nCode определяет, что нужно сделать с веткой дочерних записей, и может содержать одно из следующих значений:

TVE_COLLAPSE	сворачивание поддерева записей
TVE_COLLAPSERESET	сворачивание поддерева с одновременным удалением всех дочерних записей
TVE_EXPAND	разворачивание поддерева
TVE_TOGGLE	разворачивание поддерева, если оно свернуто, и сворачивание, если развернуто

Следующая функция только выбирает запись:

HTREEITEM CTreeCtrl::**Select**(
HTREEITEM hItem,
UINT nCode) —

осуществляет выбор записи с одновременной прокруткой просмотра дерева так, чтобы запись стала видимой. Параметр nCode определяет действия, которые следует произвести помимо собственно выбора записи, возможны следующие значения для этого параметра:

TVGN_CARET	установить выбор заданной записи
TVGN_DROPHILITE	перерисовать запись как запись назначения операции перетаскивания (drag-and-drop)
TVGN_FIRSTVISIBLE	прокрутить записи так, чтобы заданная запись стала видимой

Выполняемые данной функцией операции могут также выполняться следующей парой функций:

HTREEITEM CTreeCtrl::**SelectItem**(HTREEITEM hItem) —

устанавливает выделение заданной параметром hItem записи просмотра дерева.

HTREEITEM CTreeCtrl::**SelectDropTarget**(HTREEITEM hItem) —

перерисовывает заданную запись как запись назначения операции перетаскивания.

Поиск и сортировка записей дерева

Функции этой группы предназначены для поиска записей по различным критериям, но в отличие от последовательных списков, для просмотра дерева, прежде всего, поиск осуществляется по тому, как искомая запись соотносится с другими.

HTREEITEM CTreeCtrl::**GetNextItem**(
HTREEITEM hItem,
UINT nCode) —
возвращает дескриптор следующей записи, подходящей по заданным параметрам, или NULL, если таковая не найдена. Параметр hItem определяет дескриптор записи, относительно которой производится поиск. Параметр nCode задает критерии поиска и может содержать один из следующих параметров:

TVGN_CARET	вернуть текущую выбранную запись
TVGN_CHILD	вернуть первую дочернюю запись; в этом случае параметр *hItem* должен быть равен NULL
TVGN_DROPHILITE	вернуть запись, которая является конечной целью при операции перетаскивания записи просмотра дерева
TVGN_FIRSTVISIBLE	вернуть первую видимую запись
TVGN_NEXT	вернуть следующую запись, которая лежит на том же уровне, что и запись, относительно которой производится поиск
TVGN_NEXTVISIBLE	вернуть видимую запись, следующую за данной записью
TVGN_PARENT	вернуть родительскую запись
TVGN_PREVIOUS	вернуть предыдущую запись, которая лежит на том же уровне, что и запись, относительно которой производится поиск
TVGN_PREVIOUSVISIBLE	вернуть видимую запись, предшествующую данной записи
TVGN_ROOT	вернуть первую дочернюю запись корневой записи, поддереву которой принадлежит указанная в параметре *hItem* запись

Если нужно получить записи, лежащие на том же уровне иерархии, что и данная запись, то можно воспользоваться следующими функциями:

HTREEITEM CTreeCtrl::**GetNextSiblingItem**(HTREEITEM hItem) —
возвращает дескриптор следующей записи.

HTREEITEM CTreeCtrl::**GetPrevSiblingItem**(HTREEITEM hItem) —
возвращает дескриптор предыдущей записи.

Следующая функция возвращает дескриптор родительской записи заданной параметром *hItem* записи:

HTREEITEM CTreeCtrl::**GetParentItem**(HTREEITEM hItem)

Следующие несколько функций связаны с поиском видимых в окне просмотра дерева записей:

BOOL CTreeCtrl::**EnsureVisible**(HTREEITEM hItem) —
гарантирует, что указанная в hItem запись видна. Если нужно, то функция разворачивает необходимую ветку записей и прокручивает записи так, чтобы указанная запись была видна.

HTREEITEM CTreeCtrl::**GetFirstVisibleItem**() —
возвращает дескриптор первой видимой записи.

HTREEITEM CTreeCtrl::**GetNextVisibleItem**(HTREEITEM hItem) —
возвращает дескриптор следующей и предыдущей видимой записи относительно заданной в параметре hItem.

HTREEITEM CTreeCtrl::**GetPrevVisibleItem**(HTREEITEM hItem) —
возвращает дескриптор предыдущей видимой записи относительно заданной в параметре hItem.

HTREEITEM CTreeCtrl::**GetSelectedItem**()
возвращает дескриптор текущей выбранной записи дерева.

HTREEITEM CTreeCtrl::**GetDropHilightItem**()
возвращает дескриптор записи назначения при операции перетаскивания.

HTREEITEM CTreeCtrl::**GetRootItem**()
возвращает дескриптор корневой записи просмотра дерева.

Класс *CTreeCtrl* позволяет определить запись по координатам, и, если есть необходимость, узнать, на какой записи находится курсор мыши, следует воспользоваться функцией *HitTest*, например, в обработчике нажатия ее кнопки:

HTREEITEM CTreeCtrl::**HitTest**(
CPoint pt,
UINT* pFlags)

или

HTREEITEM CTreeCtrl::**HitTest**(TV_HITTESTINFO* pHitTestInfo) —
возвращает дескриптор записи, которой принадлежит указанная точка, или NULL, если точка не принадлежит ни одной записи. Параметр pFlags первой версии функции является указателем на переменную целого типа, принимающую результаты проверки. Может иметь одно или больше значений, указанных в описании параметра flags структуры *TV_HITTESTINFO*. Параметр pHitTestInfo является указателем на структуру типа *TV_HITTESTINFO*, содержащую координаты точки для проверки и принимающую результаты проверки.

Приведем описание этой структуры:

```
typedef struct _TVHITTESTINFO {
  POINT pt;
  UINT flags;
  HTREEITEM hItem;
} TV_HITTESTINFO;
```

Поле *flags* используется для хранения результатов проверки и может содержать комбинацию следующих значений:

TVHT_ABOVE проверяемая точка расположена выше клиентской области элемента управления

TVHT_BELOW	проверяемая точка расположена ниже клиентской области элемента управления
TVHT_NOWHERE	проверяемая точка расположена в клиентской области просмотра дерева, но лежит не в области собственно дерева
TVHT_ONITEM	проверяемая точка расположена в границах записи
TVHT_ONITEMBUTTON	проверяемая точка расположена в границах кнопки записи
TVHT_ONITEMICON	проверяемая точка расположена в границах пиктограммы записи
TVHT_ONITEMINDENT	проверяемая точка расположена в границах отступа записи
TVHT_ONITEMRIGHT	проверяемая точка расположена правее надписи записи
TVHT_ONITEMSTATEICON	проверяемая точка расположена в границах пиктограммы *состояния записи*
TVHT_TOLEFT	проверяемая точка расположена левее клиентской области элемента управления
TVHT_TORIGHT	проверяемая точка расположена правее клиентской области элемента управления

К этой же группе мы отнесли функции, осуществляющие сортировку записей просмотра дерева. Их две. Первая предназначена для сортировки по алфавиту дочерних записей родительской записи. Это функция *SortChildren*. Она определяет уровни дочерних записей, которые должны быть отсортированы:

BOOL CTreeCtrl::**SortChildren**(HTREEITEM hItem) —
сортирует дочерние записи родительской записи, указанной в hItem. Если значение параметра hItem равно NULL, то будет отсортирован весь просмотр дерева.

Если же необходимо задать собственный критерий для сортировки записей, то позволяет это сделать функция *SortChildrenCB*:

BOOL CTreeCtrl::**SortChildrenCB**(LPTV_SORTCB pSort) —
функция сортирует дочерние записи, используя определяемую приложением функцию сортировки. Параметр pSort является указателем на структуру *TV_SORTCB*:

```
typedef struct _TV_SORTCB {
  HTREEITEM hParent;
  PFNTVCOMPARE lpfnCompare;
  LPARAM lParam;
} TV_SORTCB;
```

Поле *lpfnCompare* является указателем на определяемую приложением функцию сортировки записей. Она вызывается всякий раз, когда необходимо сравнить две записи и имеет описание:

```
int CALLBACK CompareFunc(
  LPARAM lParam1,
  LPARAM lParam2,
  LPARAM lParamSort)
```

Функция должна возвращать отрицательное значение, если первая запись должна предшествовать второй, положительное, если первая запись должна следовать за второй, и нуль, если записи равны. Параметры функции сортировки аналогичны параметрам функции сортировки просмотра списка, которая была описана в соответствующем разделе.

Обработка уведомлений

В то время, когда пользователь работает с просмотром дерева, он посылает различные уведомления. Если необходимо, то для их обработки можно назначить функции-обработчики. Перечень уведомляющих сообщений, посылаемых просмотром дерева, приведен ниже:

TVN_BEGINDRAG	сигнализирует о начале операции перемещения при помощи левой кнопки мыши (drag-and-drop)
TVN_BEGINLABELEDIT	сигнализирует о начале операции редактирования надписи записи
TVN_BEGINRDRAG	сигнализирует о начале операции перемещения при помощи правой кнопки мыши (drag-and-drop)
TVN_DELETEITEM	сигнализирует об удалении конкретной записи просмотра дерева
TVN_ENDLABELEDIT	сигнализирует о завершении операции редактирования надписи записи
TVN_GETDISPINFO	запрашивает информацию о записи по запросу просмотра дерева для ее отображения
TVN_ITEMEXPANDED	сигнализирует о смене состояния записи просмотра дерева (сворачивании или разворачивании)
TVN_ITEMEXPANDING	запрашивает подтверждение смены состояния записи просмотра дерева (сворачивании или разворачивании)
TVN_KEYDOWN	сигнализирует о нажатии клавиши на клавиатуре
TVN_SELCHANGED	сигнализирует о смене выбора записи просмотра дерева
TVN_SELCHANGING	запрашивает подтверждение смены выбора записи просмотра дерева
TVN_SETDISPINFO	уведомляет родительское окно о необходимости изменения информации о записи по запросу, которая предоставляется приложением

Просмотр дерева посылает уведомления в форме сообщения WM_NOTIFY:

```
BOOL CMarsWnd::OnNotify(
  WPARAM wParam, LPARAM lParam, LRESULT* pResult)
```

```
{
  ...
  TV_DISPINFO *ptvdi = (TV_DISPINFO *)lParam;
  TV_ITEM item = ptvdi->item;
  ...
}
```

Некоторые уведомления работают с другой структурой, а именно со структурой *NM_TREEVIEW*:

```
NM_TREEVIEW *pnmtv = (NM_TREEVIEW *)lParam;
```

Механизм обработки уведомлений в принципе такой же, как и для просмотров списка, рассмотренный в предыдущем разделе.

Так же, как и класс *CListCtrl* для просмотров списка, класс *CTreeCtrl* имеет в своем составе функции поддержки редактирования надписей записи *GetEditControl* и *EditLabel*, а также функцию *CreateDragImage*, создающую временный список изображений для реализации механизма перемещения записей просмотра дерева.

Часть X

Архитектура "документ/представление"

26. Основные положения
27. Документ и его представления
28. Печать и предварительный просмотр документов

Глава 26

Основные положения

Прежде, чем переходить к рассмотрению темы, вынесенной в заголовок части, сделаем обзор тех основных возможностей, которыми уже можно осознанно пользоваться.

Каждое приложение, написанное для Windows, имеет одно главное окно, которое состоит из двух компонентов — фрейма и клиентской (рабочей) области. Оно может быть как стандартным окном Windows (например, при построении SDI- и MDI-приложений), так и обычным блоком диалога. В любом случае, внутри главного окна, приложение предоставляет пользователю всю необходимую для работы информацию, под которой понимаются не только данные, т. е. текст, рисунки и т. п., но и элементы интерфейса — меню, полосы прокрутки, панели инструментов, строка состояния и т. д. Каждый из этих элементов (по крайней мере, для приложений, созданных с использованием библиотеки классов MFC) является объектом, в состав которого входят функции, определяющие его поведение — взаимодействие как с другими объектами, так и с данными, с которыми он работает. Любые действия пользователя во время выполнения приложения вызывают генерацию сообщений, которые содержат в себе информацию о том, что конкретно сделал пользователь. Все объекты приложения обладают способностью реагировать на те или иные сообщения. Во многих случаях эта способность заложена в библиотеке классов MFC, а для других реализуются специальные обработчики. Повторим — все перечисленные свойства характерны (с той или иной степенью полноты) для каждого Windows-приложения.

Оставим в стороне приложения, базирующиеся на блоках диалога, и сосредоточимся на тех, которые в качестве своего главного окна имеют стандартное окно Windows. Основным для таких окон является объект, созданный на базе класса *CFrameWnd* (для SDI-приложений) или *CMDIFrameWnd* (для MDI-приложений). То, что делалось до сих пор, заключалось в создании некоторых необходимых оконных объектов и *самостоятельной* организации взаимодействия между ними. Мы полностью (более того — сознательно) игнорировали такое мощное средство, заложенное разработчиками в библиотеку классов MFC, как архитектура "документ/представление" (document/view).

В основе этой архитектуры лежат три глобальных понятия — *фрейм*, *документ* и *представление*. И если с фреймом мы уже достаточно близко ознакомились (некоторые его свойства, связанные с архитектурой "документ/представление", будут рассмотрены чуть ниже), то два других необходимо рассмотреть подробно, что мы и сделаем в следующей главе. Здесь же просто определим их место в архитектуре "документ/представление".

Что же включают в себя эти понятия? В самом общем случае, под *документом* фирма Microsoft понимает те данные, с которыми работает приложение. Это может быть все, что угодно, — "простой" текст, картинка и т. п. Отображение этих данных на экране осуществляется в так называемом *фрейме* документа. Разработчики библиотеки MFC создали для фреймов специальные классы окон — *представления*, которые отображают данные документа и управляют взаимодействием пользователя с ними. Другими словами, способ хранения данных в памяти или на диске никоим образом не влияет на их внешнее представление пользователю.[1]

Роль фрейма в этой архитектуре заключается в создании соответствующих фреймов документов, содержащих специальные окна — представления. Кроме того, фрейм осуществляет координацию взаимодействия между документами различных типов и их представлениями, направляя им команды и получая от них извещения. Маршруты, по которым осуществляется передача сообщений, подробно описаны нами в четвертой части.

Архитектура "документ/представление" разделяет между собой данные-документы, и их изображения. Каждый документ представлен в виде объекта, который обеспечивает пространство для хранения данных в памяти и отвечает за такие операции, как запись и чтение документа с диска. Очевидно, что большая часть таких операций выполняется самим приложением, а не классами библиотеки MFC. Кроме того, для каждого используемого представления также создается отдельный объект, который представляет собой экранное окно, отвечающее за взаимодействие с пользователем и соответствующим объектом документа, печать и т. д. Совместную работу перечисленных объектов координирует фрейм, который, в большинстве случаев, является главным окном приложения (рис. 31).

Когда запускается приложение под Windows, пользователь взаимодействует с документами посредством их изображений во фреймах. Фреймы документов имеют два основных компонента: собственно фрейм и его содержимое. Для них библиотека MFC использует два различных класса. Класс фрейма управляет непосредственно фреймом, а класс представления — его содержимым. При этом окно представления является дочерним по отношению к фрейму, т. е. размещается в его рабочей области. Сам фрейм документа может быть окном SDI-приложения или дочерним окном MDI-приложения.

---

[1] К сожалению, сама фирма Microsoft сплошь и рядом нарушает это положение, что наглядно видно на поставляемых примерах. Да и класс *CEditView* спроектирован отнюдь не на этом принципе. Мы, тем не менее, будем придерживаться этого основного положения.

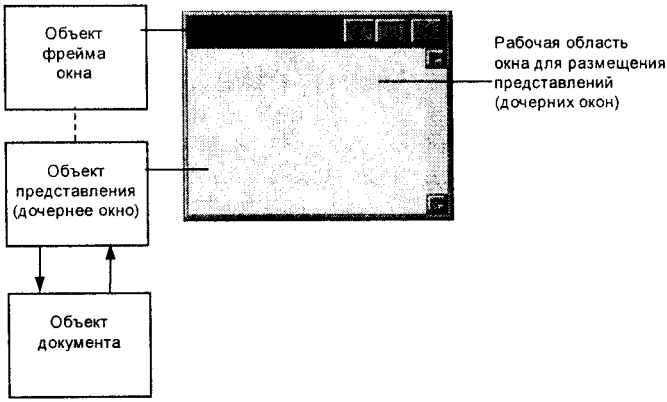


Рис. 31. Взаимосвязь между фреймом, документом и представлением

Работа с основными компонентами фрейма документа осуществляется также по-разному. За взаимодействие пользователя непосредственно с фреймом (изменение размеров, перемещение и т. д.) полностью отвечает сама система Windows. Управление же содержимым фрейма, т. е. представлениями, ложится на плечи разработчика.

Для координации создания и взаимодействия трех основных объектов рассматриваемой архитектуры в библиотеке классов MFC используются специальные классы шаблонов документов, которые создаются и управляются объектом "приложение".

Таким образом, архитектура "документ/представление" охватывает следующие основные классы:[1]

- *CWinApp* — класс для создания единственного объекта — приложения;
- *CFrameWnd* — класс для создания главного окна однодокументного приложения и базовый для классов *CMDIFrameWnd* и *CMDIChildWnd*, которые отвечают за работу многодокументного приложения;
- *CDocTemplate* — базовый абстрактный класс для создания шаблонов документов; при работе с однодокументным приложением используется производный от него класс *CSingleDocTemplate*, а для многодокументных — класс *CMultiDocTemplate*;
- *CDocument* — класс для создания собственно документа;
- *CView* — базовый класс, который совместно со своими производными классами — *CCtrlView*, *CEditView*, *CListView*, *CTreeView* и *CScrollView* — отвечает за отображение данных документа и за взаимодействие с пользователем; для этих же целей можно использовать класс *CSplitterWnd*.

Нам осталось рассмотреть вопросы их создания и взаимодействия.

---

[1] В представленный перечень не включены классы, предназначенные для работы с базами данных и OLE.

Место объекта-приложения в архитектуре "документ/представление"

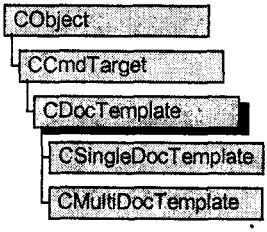
Класс *CWinApp* был нами достаточно подробно рассмотрен в первом томе, и нет смысла повторяться. Однако при этом мы не касались его свойств, связанных с архитектурой "документ/представление", отложив обсуждение этого вопроса до более удобного момента. Теперь пришло время заполнить этот пробел.

Но прежде чем обсуждать роль класса *CWinApp* в рассматриваемой архитектуре, необходимо ввести понятие шаблона документа.

Приложение может поддерживать произвольное число типов документов. И единственное, что для этого нужно — создать и "зарегистрировать" во время инициализации объекта-приложения (при выполнении функции *InitInstance*) необходимое число шаблонов документов. Причем для каждого типа документа, с которым предполагается работа, используется свой шаблон. Например, если приложение поддерживает документы двух разных типов — табличные и текстовые, то оно должно создать для них два разных шаблона.

В библиотеке классов MFC для работы с шаблонами документов реализованы специальный класс — *CDocTemplate* и два производных от него — *CSingleDocTemplate* и *CMultiDocTemplate*. Рассмотрим возможности, которые в них заложены.

Класс CDocTemplate



Это абстрактный класс, в котором реализованы основные функциональные возможности для работы с шаблонами документов — организация и управление взаимодействием между классами трех типов:

- классом документа, образованного из *CDocument*,
- классом представления, который изображает данные документа. Можно создать этот класс на базе *CView*, *CScrollView*, *CFormView*, *CEditView* и т. д. (*CEditView* можно использовать непосредственно для текстовых документов),
- классом фрейма, который содержит представление. Для SDI-приложений он создается на основе *CFrameWnd*, а для MDI-приложений — на базе *CMDIChildWnd*. Базовые классы можно использовать и непосредственно.

Рассмотрим основные компоненты этого класса. Начнем, как обычно, с конструктора, который фиксирует типы документа, фрейма и представления, доступные для работы с данным шаблоном.

CDocTemplate::**CDocTemplate**(
UINT nIDResource,
CRuntimeClass *pDocClass,
CRuntimeClass *pFrameClass,
CRuntimeClass *pViewClass) —

создает объект класса. В качестве аргументов используются: nIDResource — идентификатор ресурсов (меню, акселераторы и строка описания типа шаблона документа), используемых с этим типом документов; pDocClass — указатель на объект CRuntimeClass, отвечающий за данные документа; pFrameClass — указатель на объект CRuntimeClass, характеризующий фрейм документа; pViewClass — указатель на объект CRuntimeClass, отвечающий за представление документа. Эти объекты должны быть уже определены к моменту создания объекта "шаблон документа". Память под этот объект следует распределять динамически, и полученный указатель передать в качестве параметра в функцию CWinApp::AddDocTemplate.

virtual void CDocTemplate::**AddDocument**(CDocument *pDoc),

virtual void CDocTemplate::**RemoveDocument**(CDocument *pDoc) —

добавляет (или удаляет) документ, на который указывает pDoc, в список документов, ассоциированных с этим шаблоном. В производных классах CSingleDocTemplate и CMultiDocTemplate эти функции переопределены.

virtual void CDocTemplate::**LoadTemplate**() —

загружает ресурсы для данного шаблона документа. Обычно вызывается библиотекой классов во время создания объекта и практически не требует прямого вызова. Исключение составляет случай, когда объект класса создается на глобальном уровне. В этом случае во время выполнения функции CWinApp::AddDocTemplate необходим ее явный вызов. В качестве ресурсов, используемых документами этого типа, могут использоваться меню, акселераторы и единственный строковый ресурс.

Рассмотрим структуру строки описания типа более подробно

```
// Строка описания типа шаблона документа
STRINGTABLE PRELOAD DISCARDABLE
BEGIN
 IDR_NOTETYPE "\nNote\nNote\nNote Files (*.tnd)\n.tnd
               \nNote.Document\nNote Document"
END
```

Как видите, он состоит из семи подстрок (первая — пустая), разделенных символом "\n", которые содержат дополнительную информацию о типе документа:[1]

- Заголовок главного окна SDI-приложения (Main frame caption). Не используется для MDI-приложений;

[1] В скобках указано название соответствующего поля панели Advanced Options (Дополнительные параметры) мастера создания каркаса приложений AppWizard.

- Имя файла, присваиваемое новому документу по умолчанию (Doc type name) — *Note*. Если эта подстрока отсутствует, то используется имя *Untitled* или его аналог для локализованных версий Windows (*БезИмени* — для русской версии);
- Имя типа документа (File new name (OLE short name)) — *Note*. Если в приложении определено больше одного шаблона, то при создании нового документа на экране отображается блок диалога для выбора типа создаваемого документа;
- Описание типа документа и его фильтра (Filter name) — *Note Files (*.tnd)*. Эта строка отображается в списке типов документов стандартных блоков диалога Open (Открыть) и Save As (Сохранить как);
- Расширение, используемое для документов этого типа (File extension) — *.tnd*;
- Идентификатор типа документа, хранящийся в реестре Windows (File type ID), — *Note.Document*;
- Имя типа документа, хранящееся в реестре Windows (File new name (OLE long name)), — *Note Document*. Это же имя используется в качестве длинного имени объекта OLE, если в приложение включена соответствующая поддержка.

Если какая-либо из этих подстрок не включается в строку, то вместо нее обязательно используется разделяющий символ "\n". Заключительный символ "\n" в строке является необязательным.

virtual BOOL CDocTemplate::**GetDocString**(
CString& rString,
enum DocStringIndex index) —

позволяет получить определенную подстроку описывающую тип документа, которая хранится в шаблоне и получена из соответствующей строки файла ресурсов приложения. Найденная подстрока записывается в rString. Параметр index определяет индекс подстроки, информацию о котором необходимо получить. Он может принимать одно из следующих значений:

CDocTemplate::**windowTitle**	имя, отображаемое в заголовке главного окна приложения; представлено только в шаблоне документа для SDI-приложений.
CDocTemplate::**docName**	основа имени документа; имя нового документа этого типа складывается из основы, к которой добавляется число; если значение не определено, то по умолчанию используется *Untitled* (для русской версии Windows – *БезИмени*).
CDocTemplate::**fileNewName**	имя типа этого документа; если значение не определено, то этот тип документа недоступен при выполнении команды File\New (Файл\Создать).

CDocTemplate::**filterNewName**	описание типа документа и нейтрального символа для фильтра, сопоставляемого с документами этого типа; эта строка изображается в списке типов файлов в блоке диалога File Open (Открыть файл); если значение не определено, то этот тип документа недоступен при выполнении команды File\Open (Файл\Открыть).
CDocTemplate::**filterExt**	расширение документов этого типа; если значение не определено, то этот тип документа недоступен при выполнении команды File\Open (Файл\Открыть).
CDocTemplate::**regFileTypeId**	идентификатор типа документа, хранящийся в реестре Windows; эта строка только для внутреннего использования; если не определена, то тип документа не может быть зарегистрирован в File Manager (Диспетчер файлов).
CDocTemplate::**regFileTypeName**	имя типа документа, хранящегося в регистрационной базе данных; эта строка может быть отображена в блоках диалога приложения, вызываемых для доступа к регистрационной базе данных.

virtual BOOL CDocTemplate::**SaveAllModified**() —
позволяет сохранить все модифицированные документы, которые ассоциированы с этим шаблоном.

virtual void CDocTemplate::**CloseAllDocuments**(BOOL bEndSession) —
позволяет закрыть все открытые документы. Обычно используется как часть кода, обрабатывающего команду Exit (Выход). Реализация по умолчанию вызывает функцию CDocument::DeleteContents, которая удаляет данные из документов и закрывает фреймы всех представлений, присоединенных к документу. Параметр bEndSession определяет, нужно ли завершать сеанс работы (TRUE) или нет (FALSE). Если перед закрытием документа необходимо выполнить какие-либо специфические действия, то функцию следует переопределить.

Кроме вышеперечисленных, в классе объявлены еще четыре "чистые" функции: *GetFirstDocPosition*, *GetNextDoc*, *OpenFileName* и *SetDefaultTitle*. Наличие этих функций превращает *CDocTemplate* в абстрактный класс и, следовательно, его нельзя использовать непосредственно. Обычно приложения используют один из двух производных от него классов, предоставляемых библиотекой MFC: *CSingleDocTemplate* — для SDI- и *CMultiDocTemplate* — для MDI-приложений.

Если по каким-либо причинам может потребоваться интерфейс, который фундаментально отличается от SDI и MDI, то можно образовать свой класс непосредственно из *CDocTemplate*.

Для полноты картины приведем прототипы еще нескольких функций этого класса, которые в явном виде используются достаточно редко, но в некоторых случаях могут быть полезны.

Если шаблон поддерживает несколько типов файлов, то, используя функцию *MatchDocType* для каждого шаблона, можно определить, какие шаблоны документов подходят для данного файла, и выбрать тот, который нужен:

virtual Confidence CDocTemplate::**MatchDocType**(
LPCTSTR lpszPathName,
CDocument *&rpDocMatch) —

служит для определения типа шаблона документа, используемого для открытия файла. Параметры: lpszPathName — полное имя файла; rpDocMatch — указатель на уже открытый документ, если файл, определенный в lpszPathName, уже открыт. Функция возвращает значение, определенное в перечислении *Confidence*:

```
enum Confidence
{
  noAttempt,
  maybeAttemptForeign,
  maybeAttemptNative,
  yesbeAttemptForeign,
  yesbeAttemptNative,
  yesAlreadyOpen
};
```

Если файл, определенный в параметре lpszPathName, уже открыт, то функция возвращает CDocTemplate::yesAlreadyOpen и копирует объект CDocument файла в объект rpDocMatch. Если файл еще не открыт, но расширение имени в lpszPathName совпадает со значением, определенным в CDocTemplate::filterExt, то функция возвращает CDocTemplate::yesAttemptNative и устанавливает rpDocMatch в NULL. В остальных случаях функция возвращает значение CDocTemplate::yesAttemptForeign. Другие члены перечисления можно использовать при переопределении этой функции.

virtual CDocument* CDocTemplate::**CreateNewDocument**() —

создает новый документ, тип которого определяется текущим шаблоном. Функция возвращает указатель на вновь созданный документ или NULL в случае ошибки.

virtual CFrameWnd* CDocTemplate::**CreateNewFrame**(
CDocument *pDoc,
CFrameWnd *pOther) —

создает новый фрейм (обычно невидимый) с присоединенными к нему документом и представлением. Параметр pDoc указывает на документ, на который ссылается новый фрейм. Если он равен NULL, то библиотека выводит соответствующее информационное сообщение. При создании нового фрейма можно использовать модель, которая задается параметром pOther. Эта функция позволяет создать новый фрейм вне стандартной реализации обработчиков команд ID_FILE_NEW и ID_FILE_OPEN. Она возвращает указатель на вновь созданный фрейм или NULL, если произошла какая-либо ошибка.

virtual void CDocTemplate::**InitialUpdateFrame**(
CFrameWnd *pFrame,
CDocument *pDoc,
BOOL bMakeVisible = TRUE) –

применяется совместно с функцией CreateNewFrame для создания нового фрейма вне стандартной реализации команд ID_FILE_NEW и ID_FILE_OPEN и служит для получения представлением фрейма (параметр pFrame) вызова функции *OnInitialUpdate*. Параметр pDoc указывает на документ, ассоциированный с фреймом, или содержит NULL. Параметр *bMakeVisible* показывает, должен ли фрейм стать видимым и активным.

Как мы уже сказали, *CDocTemplate* – абстрактный класс, и на его основе объекты шаблонов не создаются. Его основная роль заключается в обеспечении шаблонов основными свойствами, а для создания конкретных объектов в библиотеке классов MFC реализованы два класса – *CSingleDocTemplate* и *CMultiDocTemplate*, которые специально адаптированы для работы соответственно в SDI- и MDI-приложениях.

Класс CSingleDocTemplate

Данный класс определяет шаблон документа для однодокументного интерфейса. В SDI-приложениях главный фрейм является одновременно и фреймом документа, т. е. в конкретный момент времени может быть открыт только один документ. Обычно SDI-приложения поддерживают один тип документа (хотя это и не является обязательным условием), т. е. они содержат только один объект класса *CSingleDocTemplate*.

Объекты этого класса обрабатываются внутри библиотеки MFC. В своей защищенной части класс содержит член, в котором хранится указатель на документ, присоединенный к шаблону:

```
class CSingleDocTemplate : public CDocTemplate
{
...
protected:
  CDocument *m_pOnlyDoc;
};
```

Ниже представлены основные функции класса.

CSingleDocTemplate::**CSingleDocTemplate**(
UINT nIDResource,
CRuntimeClass *pDocClass,
CRuntimeClass *pFrameClass,
CRuntimeClass *pViewClass) –

создает объект соответствующего класса. Назначение параметров приведено в описании конструктора класса CDocTemplate.

Помимо конструктора, в классе реализованы все четыре "чистые" функции класса *CDocTemplate*.

virtual POSITION CDocTemplate::**GetFirstDocPosition**() —
возвращает позицию первого документа из списка документов, ассоциированных с данным шаблоном, или NULL, если список пуст. Она переопределена в обоих производных классах — CSingleDocTemplate и CMultiDocTemplate. Если возникло желание создать свой класс на базе CDocTemplate, то эту функцию следует в нем переопределить.

virtual CDocument* CDocTemplate::**GeNextDoc**(POSITION &rPos) —
возвращает указатель на объект-документ из общего списка документов, ассоциированных с шаблоном, который хранится непосредственно за документом, заданным его позицией в списке (параметр rPos). Позиция полученного документа записывается в rPos. Если возвращен указатель на последний элемент списка, то в параметр rPos функция записывает NULL. Для получения указателя на первый документ списка необходимо вызвать эту функцию до GetFirstDocPosition. При использовании в качестве параметра rPos недопустимого значения библиотека MFC генерирует исключение.

virtual CDocument* CDocTemplate::**OpenDocumentFile**(
LPCTSTR lpszPathName,
BOOL bMakeVisible = TRUE) —
позволяет открыть файл, заданный полным именем lpszPathName, который содержит необходимый документ. Если значение этого параметра равно NULL, то вызывается функция CreateNewDocument, которая создает новый файл, содержащий документ типа, ассоциированного с этим шаблоном. При успешном завершении функция возвращает указатель на документ, в противном случае NULL.

virtual void CDocTemplate::**SetDefaultTitle**(CDocument *pDoc) —
загружает заголовок документа, используемый по умолчанию и определенный в строке ресурса, и выводит его в полосу заголовка фрейма документа.

Кроме перечисленных "чистых" функций, в классе переопределены также функции базового класса *AddDocument* и *RemoveDocument*.

Класс CMultiDocTemplate

Этот класс определяет шаблон документа для многодокументного интерфейса. MDI-приложения используют главный фрейм как рабочее место, в котором пользователь может открывать произвольное число окон документа, отображающих данные того или иного документа. Они могут поддерживать больше одного шаблона документов, и документы различных типов могут быть открыты одновременно.

Приложение использует шаблон(ы) документа, когда пользователь создает новый документ. Если приложение поддерживает больше одного типа документов, то библиотека получает имена поддерживаемых типов из шаблона и отображает их в списке блока диалога New, реализованного на базе класса

CNewTypeDlg. Как только пользователь выбрал тип документа, приложение создает объекты "документ", "фрейм" и "представление" и сопоставляет их друг другу.

В классе определены два общедоступных члена, отвечающих за совместное использование меню и таблицы акселераторов:

```
HMENU  CMultiDocTemplate::m_hMenuShared;
HACCEL CMultiDocTemplate::m_hAccelTable;
```

Поскольку в многодокументных приложениях одновременно могут быть открыты несколько документов, в классе предусмотрен защищенный член:

```
CPtrList m_docList;
```

хранящий список открытых документов этого типа. Доступ к элементам списка осуществляется посредством функций *GetFirstDocPosition* и *GetNextDoc*.

Число непоименованных открытых документов хранится в специальном защищенном члене класса (нумерация начинается с нуля):

```
UINT m_nUntitledCount;
```

Значение, хранящееся в этой переменной, используется для генерации имени по умолчанию, и автоматически *увеличивается* при создании новых документов.

В MDI-приложениях определен, прежде всего, конструктор класса.

CMultiDocTemplate::**CMultiDocTemplate**(
  UINT nIDResource,
  CRuntimeClass *pDocClass,
  CRuntimeClass *pFrameClass,
  CRuntimeClass *pViewClass) —
    создает объект класса.

Помимо конструктора, в классе реализованы также все четыре "чистые" функции базового класса: *GetFirstDocPosition*, *GetNextDoc*, *OpenFileName* и *SetDefaultTitle*, описание которых приведено выше. Кроме того, "наполнены новым содержанием" еще три функции: *LoadTemplate*, *AddDocument* и *RemoveDocument*.

В заключение обсуждения вопросов создания и использования шаблонов документов рассмотрим, каким образом с документом ассоциируется некоторая пиктограмма. Начнем с того, что написано в документации. "Пиктограмма, зарегистрированная для каждого шаблона документа, базируется на его позиции в списке шаблонов приложения. Порядок расположения шаблонов определяется порядком его занесения в список, т. е. порядком вызова функций *AddDocTemplate*. При этом библиотека MFC присваивает первый

ресурс пиктограммы самому приложению, следующий — первому документу, и т. д.". А теперь попытаемся разобраться, что же все-таки здесь имеется в виду, ведь при работе пиктограмма большей частью определяется своим числовым идентификатором, а отнюдь не "...позицией в списке...". Рассмотрим фрагмент файла resource.h приложения NoteDraw:

```
...
#define IDR_MAINFRAME 128 //идентификатор главного окна
#define IDR_NOTETYPE  129 //идентификатор окна текстового документа
#define IDR_DRAWTYPE  130 //идентификатор окна графического
                          //документа
#define IDI_NOTEICON  129 //идентификатор пиктограммы окна
                          //текстового документа
#define IDI_DRAWICON  130 //идентификатор пиктограммы окна
                          //графического документа
...
```

Прежде всего следует обратить внимание на то, что числа, характеризующие идентификаторы ресурсов документов и пиктограмм, принадлежат одному ряду. Это условие является обязательным, т. к. в противном случае на экране вы увидите пиктограмму по умолчанию вместо ожидаемой. И, наконец, числовые значения идентификаторов ресурсов документов и пиктограмм должны совпадать, иначе можно в окне текстового документа увидеть пиктограмму графического и наоборот. Во избежание подобных недоразумений рекомендуем для пиктограмм использовать тот же идентификатор, что и для ресурсов документов: IDR_NOTETYPE и IDR_DRAWTYPE.

Возвращаясь к классу приложения *CWinApp*, рассмотрим кратко те его функции, которые отвечают за взаимодействие с другими объектами, входящими в архитектуру "документ/представление".

void CWinApp::**AddDocTemplate**(CDocTemplate *pTemplate) —
добавляет шаблон документа pTemplate в список доступных шаблонов, которые поддерживает приложение. Все необходимые добавления в шаблон документа следует производить в функции InitInstance до вызова функции RegisterShellFileTypes, т. к. в противном случае типы документов не будут представлены в реестре.

В качестве примера рассмотрим фрагмент кода приложения NoteDraw, отвечающий за создание двух шаблонов документов, один из которых позволяет создавать документы для работы с текстом, а второй — с графическими объектами:[1]

```
...
CMultiDocTemplate* pDocTemplate;
```

[1] Полный текст приложения имеется на сопроводительной дискете.

```
// Создаем объект "шаблон документа" для работы с текстом
pDocTemplate = new CMultiDocTemplate(
                                        RUNTIME_CLASS(CNoteFrame),
                                        RUNTIME_CLASS(CNoteDoc),
                                        RUNTIME_CLASS(CNoteView));
                                        IDR_NOTETYPE,
// Добавляем шаблон в список
AddDocTemplate(pDocTemplate);

// Создаем объект "шаблон документа" для работы с графикой
pDocTemplate = new CMultiDocTemplate(IDR_DRAWTYPE,
                                        RUNTIME_CLASS(CDrawDoc),
                                        RUNTIME_CLASS(CDrawFrame),
                                        RUNTIME_CLASS(CDrawView));

// Добавляем шаблон в список
AddDocTemplate(pDocTemplate);

...

STRINGTABLE PRELOAD DISCARDABLE
BEGIN
  IDR_MAINFRAME "NoteDraw"
  IDR_NOTETYPE  "\nNote\nNote\nNote Files
                 (*.tnd)\n.tnd\nNote.Document\nNote Document"
  IDR_DRAWTYPE  "\nDraw\nDraw\nDraw Files
                 (*.dnd)\n.dnd\nDraw.Document\nDraw Document"
END

...
```

POSITION CWinApp::**GetFirstDocTemplatePosition**() —
позволяет получить позицию первого шаблона документа приложения. Если список пуст, то возвращается NULL.

CDocTemplate* CWinApp::**GetNextDocTemplate**(POSITION &pos) —
позволяет получить шаблон документа, определяемый его позицией (параметр pos) в списке шаблонов приложения. После возврата из функции в pos содержится значение позиции следующего шаблона документа. Если полученный шаблон был последним в списке, то pos устанавливается в NULL.

Использование этих двух функций позволяет просматривать список доступных шаблонов документов приложения. При этом порядок их расположения в списке определяется порядком их записи туда посредством вызова функции *AddDocTemplate*.

virtual CDocument* CWinApp::**OpenDocumentFile**(LPCTSTR lpszFileName) —
открывает файл документа, задаваемый именем lpszFileName, и возвращает указатель на соответствующий объект-документ. Если к моменту вызова функции файл уже был открыт, то активизируется фрейм, содержащий этот документ. Если приложение поддерживает несколько шаблонов документов, то для поиска соответствующего шаблона библиотека MFC использует расширение

файла. В случае успешного выполнения шаблон создает фрейм и представление соответствующего документа.

void CWinApp::**CloseAllDocuments**(BOOL bEndSession) —

закрывает все открытые документы до выхода из приложения. Если при этом параметр bEndSession равен TRUE, то завершается и сеанс работы с Windows. Функцию следует вызывать после функции HideApplication.

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

вызывается библиотекой MFC для того, чтобы сохранить все документы, когда закрывается главное окно приложения или обрабатывается сообщение WM_QUERYENDSESSION. Реализация этой функции по умолчанию вызывает функцию CDocument::SaveModified для сохранения всех изменений в документах, открытых внутри приложения.

afx_msg void CWinApp::**OnFileNew**()

и

afx_msg void CWinApp::**OnFileOpen**() —

обработчики команд ID_FILE_NEW и ID_FILE_OPEN. Если обработка команды возлагается на библиотеку классов MFC, в карту сообщений класса приложения необходимо добавить операторы

```
ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
```

или

```
ON_COMMAND(ID_FILE_NEW, OnNameCommandFileNew)
ON_COMMAND(ID_FILE_OPEN, OnNameCommandFileOpen)
```

если вы хотите самостоятельно обрабатывать эту команду. При этом на плечи программиста ложится реализация обработчика OnNameCommandFileNew (имя, естественно, произвольное).

В свете архитектуры "документ/представление" основным назначением объекта-приложения является то, что он содержит полный список шаблонов документов, используя для этого общедоступный член

CDocManager* CWinApp::**m_pDocManager**;

Фактически список хранится в защищенном члене класса библиотеки MFC — *CDocManager*

CPtrList CDocManager::**m_templateList**;

и доступ к нему осуществляется при помощи рассмотренных функций объекта-приложения *GetFirstDocTemplatePosition* и *GetNextDocTemplate*, которые после предварительной обработки передают управление соответствующей функции класса *CDocManager*. В качестве примера рассмотрим три фрагмента кода из исходных текстов реализации функций классов *CWinApp* и *CDocManager*, первый из которых отвечает за добавление документа в шаблон, а остальные обрабатывают команду ID_FILE_NEW:

```
// Первый фрагмент: добавление шаблона в список
...
// Создан объект класса CDocManager?
if (m_pDocManager == NULL)
{
  // если нет, то создаем его
  m_pDocManager = new CDocManager;
}
// и в обоих случаях передаем ему управление
// для добавления шаблона документа в общий список
m_pDocManager->AddDocTemplate(pTemplate);
...
// Второй фрагмент: обработка команды ID_FILE_NEW
// в классе CWinApp
...
// Создан объект класса CDocManager?
if (m_pDocManager != NULL)
{
  // если да, то передаем ему управление для выбора шаблона
  m_pDocManager->OnFileNew();
}
...
// Третий фрагмент: обработка команды ID_FILE_NEW
// в классе CDocManager
...
// Проверяем, имеется ли в списке какой-либо шаблон
if (m_templateList.IsEmpty())
{
  // Если список пуст, то вывод сообщения об ошибке и возврат
  ...
  return;
}
// Получаем указатель на первый в списке шаблон документов
CDocTemplate* pTemplate=(CDocTemplate*)m_templateList.GetHead();
// В списке больше одного шаблона?
if (m_templateList.GetCount() > 1)
{
  // Создание и вывод на экран блока диалога
    // со списком доступных шаблонов
  CNewTypeDlg dlg(&m_templateList);
  int nID = dlg.DoModal();
```

```
  if (nID == IDOK)
  {
    // Фиксируем выбранный шаблон
    pTemplate = dlg.m_pSelectedTemplate;
  }
  else
  {
    return;
  }
}

// Открываем новый, пустой документ на базе выбранного шаблона
pTemplate->OpenDocumentFile(NULL);

...
```

Наибольший интерес здесь представляет тот факт, что приложение может одновременно поддерживать несколько шаблонов (или типов) документов (рис. 32).

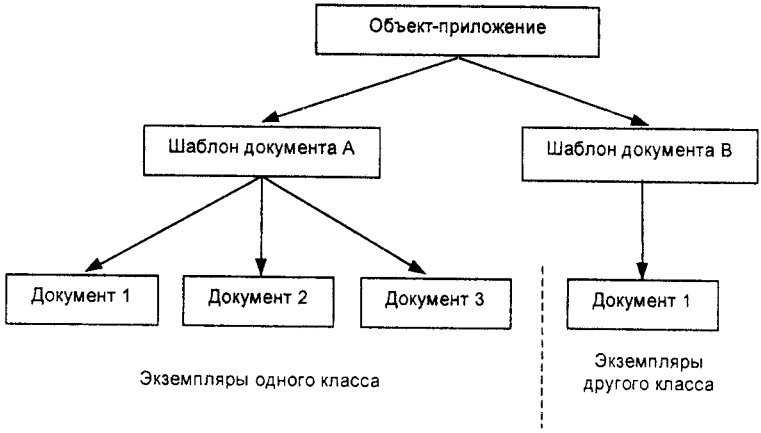


Рис. 32. Несколько документов двух типов, используемых одним приложением

В этом случае для выбора необходимого шаблона при обработке команды ID_FILE_NEW, создается и выводится на экран специальный блок диалога (рис. 33), реализованного в библиотеке MFC (файл DOCMGR.CPP) класса *CNewTypeDlg*.

```
class CNewTypeDlg : public CDialog
{
protected:
  CPtrList*     m_pList;              // список шаблонов документов

public:
  CDocTemplate* m_pSelectedTemplate; // выбранный шаблон
```

```
public:
  //{{AFX_DATA(CNewTypeDlg)
  enum { IDD = AFX_IDD_NEWTYPEDLG };
  //}}AFX_DATA

  CNewTypeDlg(CPtrList* pList) : CDialog(CNewTypeDlg::IDD)
  {
    m_pList = pList;
    m_pSelectedTemplate = NULL;
  }
...

};
```

Он позволяет пользователю просматривать список имеющихся у приложения шаблонов документов и выбирать один из них.[1]

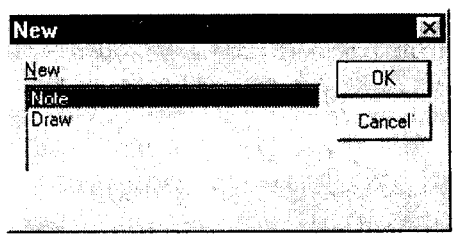


Рис. 33. Блок диалога для выбора типа вновь создаваемого документа

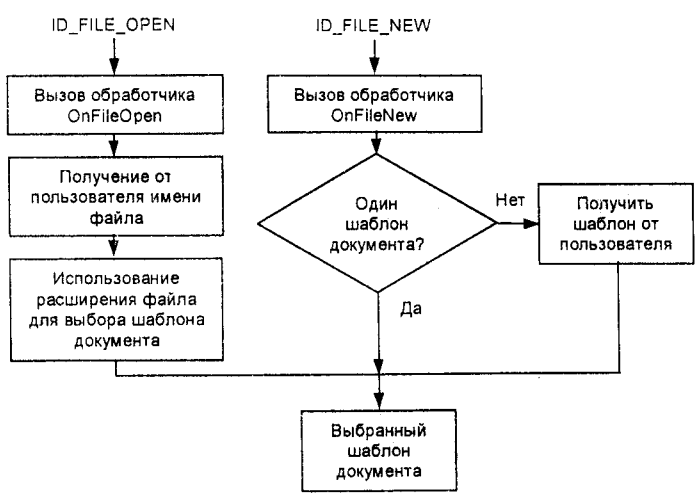


Рис. 34. Процесс создания шаблона документа

[1] В принципе, вместо этого класса можно использовать и созданный самостоятельно, но мы не видим серьезных причин для этого.

Таким образом, если внимательно посмотреть на перечисленные функции и члены класса *CWinApp*, при работе в рамках архитектуры "документ/представление", роль объекта-приложения заключается в создании, хранении и организации доступа к списку доступных для использования шаблонов документов. Фактически его роль сводится лишь к передаче управления "диспетчеру документов", представленному классом *CDocManager*, который, собственно, и выполняет всю необходимую для этого работу. Схематично процесс создания шаблона документа изображен на рис. 34.

Роль фреймов в архитектуре "документ/представление"

Фрейм документа, который мы будем здесь рассматривать, имеет две основные составляющие — собственно фрейм и его содержимое, представляющее на экране данные документа. Эти два компонента представлены и управляются различными группами классов библиотеки MFC: классы фреймов (*CFrameWnd* и *CMDIChildWnd*) и классы представлений (*CView*, *CScrollView* и другие). Процесс создания главного и дочерних фреймов подробно обсуждался в первом томе (см. главы 11, 12) и нет смысла повторяться. Вместо этого рассмотрим несколько фрагментов кода из приложения NoteDraw:

```
// описание класса (файл Childfrm.h)
class CNoteFrame : public CMDIChildWnd
{
// Для возможности динамического создания объекта
// класса CNoteFrame вызывается макрос
  DECLARE_DYNCREATE(CNoteFrame)

protected:
  CNoteFrame();

...
};
```

и

```
// реализация класса (файл Childfrm.cpp)

// Для возможности динамического создания объекта
// класса CNoteFrame вызывается макрос
IMPLEMENT_DYNCREATE(CNoteFrame, CMDIChildWnd)

...
```

Интерес здесь представляют два макроса, подробно описанные в главе 8 первого тома — DECLARE_DYNCREATE и IMPLEMENT_DYNCREATE — и защищенный (protected) конструктор по умолчанию. Дело в том, что все вышеперечисленные составляющие архитектуры "документ/представление"

(документ, фрейм и представление) создаются шаблоном документов *динамически*, во время выполнения программы. Кроме того, эти макросы охватывают также все возможности двух других групп макросов — DECLARE_DYNAMIC, IMPLEMENT_ DYNAMIC, DECLARE_SERIAL и IMPLEMENT_ SERIAL, что гарантирует также поддержку получения информации времени выполнения и сериализации. Забыв включить эти макросы соответственно в объявление и реализацию некоторого класса, вы тем самым исключаете для него возможность корректной работы механизмов RUNTIME_CLASS и сериализации.

Рассмотрим функции класса *CFrameWnd*, которые связаны с архитектурой "документ/представление".

void BOOL CFrameWnd::**OnCreateClient**(
LPCREATESTRUCT lpcs,
CCreateContext *pContext) —

вызывается только библиотекой классов MFC из обработчика OnCreate и не должна вызываться непосредственно. В качестве параметров в функцию передаются: lpcs — указатель на структуру *CREATESTRUCT*[1] и pContext — указатель на структуру CCreateContext:

```
struct CCreateContext{
  CRuntimeClass* m_pNewViewClass;
  CDocument* m_pCurrentDoc;
  CDocTemplate* m_pNewDocTemplate;
CView* m_pLastView;
  CFrameWnd* m_pCurrentFrame;
CCreateContext();
};,
```

Структура *CCreateContext* создается шаблоном документа (при создании документа и ассоциированных с ним компонентов). Перечислим поля этой структуры (все поля могут содержать NULL):

m_pNewViewClass	указатель на структуру *CRuntimeClass* вновь создаваемого представления
m_pCurrentDoc	указатель на уже существующий документ, с которым ассоциируется новое представление
m_pNewDocTemplate	указатель на шаблон документа, с которым ассоциирован создаваемый фрейм

[1] Будучи верной себе, фирма Microsoft не рекомендует вносить какие-либо изменения в поля этой структуры, т. е. предлагает считать, что она доступна "только для чтения", "любезно предоставляя" возможность это делать, например, в функции *OnCreate*.

m_pLastView — указатель на начальное представление, к которому добавляется создаваемое, например, при работе с разделенным окном или при создании дополнительного представления документа

m_pCurrentFrame — указатель на текущий фрейм, к которому добавляется создаваемый, например, при создании дополнительного фрейма документа

Библиотекой MFC эта структура используется в следующих функциях: *CWnd::Create, CFrameWnd::CreateView, CFrameWnd::LoadFrame, CFrameWnd::OnCreateClient, CFrameWnd::Create, CSplitterWnd::Create* и *CSplitterWnd::CreateView*.

virtual void CFrameWnd::**ActivateFrame**(int nCmdShow = -1) —

активизирует фрейм, т. е. делает его видимым и доступным для пользователя. Параметр nCmdShow принимает такие же значения, как и посылаемые в функцию CWnd::ShowWindow. По умолчанию функция делает фрейм видимым и переводит его на самый верх Z-порядка, и при необходимости то же самое выполняет для главного окна приложения. При переопределении, после проведения необходимых действий, например, разворачивания дочернего окна, необходимо вызвать соответствующую функцию базового класса с явным указанием параметра nCmdShow, как, например, сделано в приложении NoteDraw.

```
void CNoteFrame::ActivateFrame(int nCmdShow)
{
// Разворачиваем фрейм документа
  CMDIChildWnd::ActivateFrame(SW_MAXIMIZE);
}
```

void CFrameWnd::**InitialUpdateFrame**(
CDocument *pDoc,
BOOL bMakeVisible) —

вызывается после создания нового фрейма и инициирует вызов обработчика OnInitialUpdate для всех своих представлений. При этом активизируется представление, у которого идентификатор дочернего окна установлен в AFX_IDW_PANE_FIRST. Если значение параметра bMakeVisible равно TRUE, то фрейм делается видимым; в противном случае — нет. Параметр pDoc определяет документ, ассоциированный с данным фреймом.

virtual CFrameWnd* CFrameWnd::**GetActiveFrame**() —

возвращает указатель на активное дочернее окно MDI-приложения. При работе с SDI-приложениями функция возвращает указатель на текущий объект — *this*.

void CFrameWnd::**SetActiveView**(
CView *pViewNew,
BOOL bNotify = TRUE) —

переводит представление pView в активное состояние. Эта функция автоматически вызывается библиотекой MFC, когда пользователь сменил фокус представления внутри фрейма. Ее можно вызвать специально для программной смены фокуса представления.

```
// Установить фокус на область (1, 0)
SetActiveView((CView*)m_spltWnd.GetPane(1, 0));
```

Параметр bNotify определяет, будет ли само представление извещено об активизации. Если его значение равно TRUE, то для активизируемого представления вызывается функция OnActivateView.

CView* CFrameWnd::**GetActiveView**() —
возвращает указатель на объект текущего активного представления. Если такого нет, то возвращается NULL.

CWnd* CFrameWnd::**CreateView**(
CCreateContext *pContext,
UINT nID = AFX_IDW_PANE_FIRST) —
создает представление внутри фрейма, которое не является производным от CView, и возвращает указатель на него. После выполнения функции необходимо самостоятельно установить активность и видимость вновь созданного представления. Параметр pContext определяет тип представления и документа, а nID — идентификатор представления.

virtual CDocument* CFrameWnd::**GetActiveDocument**() —
возвращает указатель на документ, ассоциированный с текущим активным представлением. Если такого нет, что, в принципе, возможно, то функция возвращает значение NULL.

В качестве примера (фрагмент кода из приложения NoteDraw) рассмотрим, как можно переопределить функцию *OnCreateClient* для создания статического разделенного окна с двумя горизонтальными областями, в каждой из которых создается свое представление[1] (рис. 35).

```
BOOL CNoteFrame::OnCreateClient(LPCREATESTRUCT lpcs,
                                CCreateContext* pContext)
{
// создаем разделенное окно Windows
// с двумя строками и одним столбцом
  if(!m_spltWnd.CreateStatic(this, 2, 1))
  {
    ...
  }

// создаем представление в первой строке (горизонтальная область)
  if(!m_spltWnd.CreateView(0, 0, pContext->m_pNewViewClass,
      CSize(lpcs->cx, lpcs->cy*2/3), pContext))
  {
    ...
  }
```

[1] Более подробно все перечисленные действия будут описаны в следующей главе при обсуждении собственно представлений.

```
// создаем представление во второй строке (горизонтальная область)
  if(!m_spltWnd.CreateView(1, 0, RUNTIME_CLASS(CTextView),
      CSize(0, 0), pContext))
  {
    ...
  }
// устанавливаем фокус на вторую горизонтальную область
  SetActiveView((CView*)m_spltWnd.GetPane(1, 0));
  return TRUE;
}
```

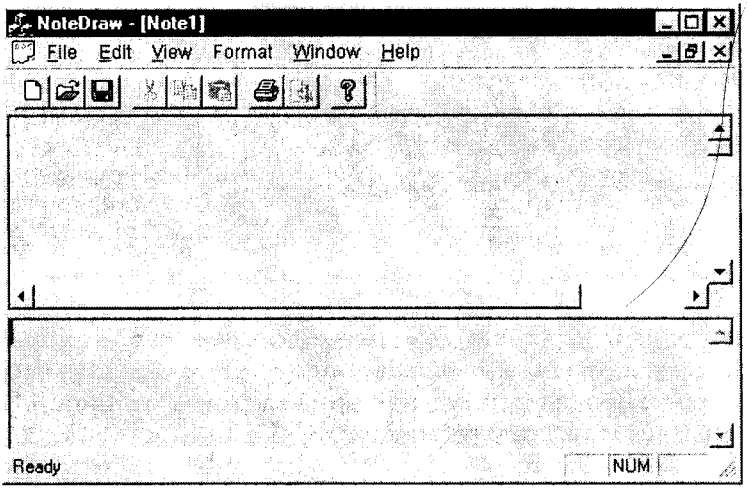


Рис. 35. Результат выполнения переопределенной функции OnCreateClient

Характер рассмотренных функций достаточно наглядно показывает роль фрейма в архитектуре "документ/представление". Прежде всего, фрейм хранит путь к текущему активному представлению. В случае, когда он содержит более одного представления, например, при работе с разделенным окном, то в качестве активного принимается самое последнее используемое представление.[1] Последняя из описанных выше функций показывает, что фрейм имеет также доступ к документу, ассоциированному с текущим активным представлением. Кроме того, фрейм занимается координацией взаимодействия между различными представлениями, ассоциированными с определенным документом, направляя им команды и получая от них извещения. Маршруты, по которым осуществляется обмен сообщениями, подробно описаны нами в главе 10. И последнее. Мы уже говорили, что с каждым

[1] Активность представления не зависит от активного окна Windows или текущего фокуса ввода.

документом ассоциируются определенные ресурсы меню и акселераторов. Так вот, задачу разделения этих ресурсов среди документов, созданных на основе одного шаблона, и замены ресурсов при активизации документа другого типа также с успехом решают главный фрейм и фрейм документа.

В заключение коротко перечислим те шаги, которые необходимо проделать для организации работы в рамках архитектуры "документ/представление":

- создать объект-приложение;
- создать объекты-документы (см. главу 27);
- создать необходимое число представлений каждого документа (см. главу 27);
- в функции *InitInstance* объекта-приложения создать объекты-шаблоны документов и добавить их к его списку;
- создать необходимые ресурсы для каждого из них;
- переопределить функцию *CFrameWnd::OnCreateClient*, в которой присоединить к фрейму документа необходимые представления;
- создать и вывести на экран главное окно приложения с присоединенными к нему фреймами документа.

Теперь, когда мы разобрались в общей структуре рассматриваемой архитектуры, перейдем к вопросам создания и работы с ее основными составляющими — документами и ассоциированными с ними представлениями.

Глава 27

Документ и его представления

Как мы уже неоднократно отмечали, одним из основных достоинств архитектуры "документ/представление" является то, что в ней данные отделены от их изображения.[1] Любые изменения данных осуществляются посредством специального класса документа, который обеспечивает необходимое пространство для их хранения в памяти, отвечает за запись и чтение документа с диска и предоставляет интерфейс для доступа и обновления данных. За изображение данных на экране, принтере или любом другом устройстве отвечает представление, которое использует для этого специальный класс *CView*, или производный от него. Можно сказать, что объект этого класса представляет собой окно, посредством которого, с одной стороны, осуществляется взаимодействие с пользователем, а с другой — организуется доступ к интерфейсной части документа для возможности обновления данных. В достаточно упрощенном виде сказанное можно проиллюстрировать с помощью рис. 36. Следует также повторить, что один и тот же документ одновременно может иметь несколько различных представлений.

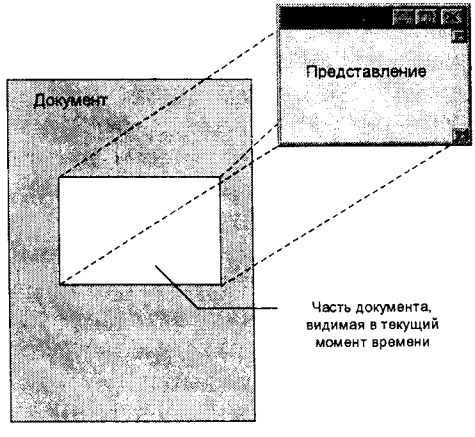


Рис. 36. Взаимоотношения документа и представления

[1] Точнее, архитектура позволяет это сделать, и мы рекомендуем к этому стремиться.

Таким образом, действуя совместно, документ и его представления

- содержат, изображают данные приложения и управляют ими;
- предоставляют интерфейс с данными документа;
- участвуют в чтении и записи файлов;
- предоставляют необходимые возможности для печати данных;
- обрабатывают большинство команд и сообщений приложения.

Рассмотрим эти основные компоненты архитектуры "документ/представление" подробно и начнем с документов.

Документы

В самом общем случае можно сказать, что документы содержат и управляют данными приложения. Или, другими словами, документ представляет собой некоторую единицу данных, которую пользователь обычно открывает по команде ID_FILE_OPEN и сохраняет по команде ID_FILE_SAVE.

Для реализации документа в типичном приложении необходимо проделать следующую последовательность действий (рис. 37):

- для каждого типа документа образовать класс на базе *CDocument;*
- добавить в него переменные для хранения всех данных документа;
- реализовать функции для чтения и модификации этих данных;
- переопределить функцию *CObject::Serialize* в новом классе документа для организации чтения/записи данных документа с диска/на диск.

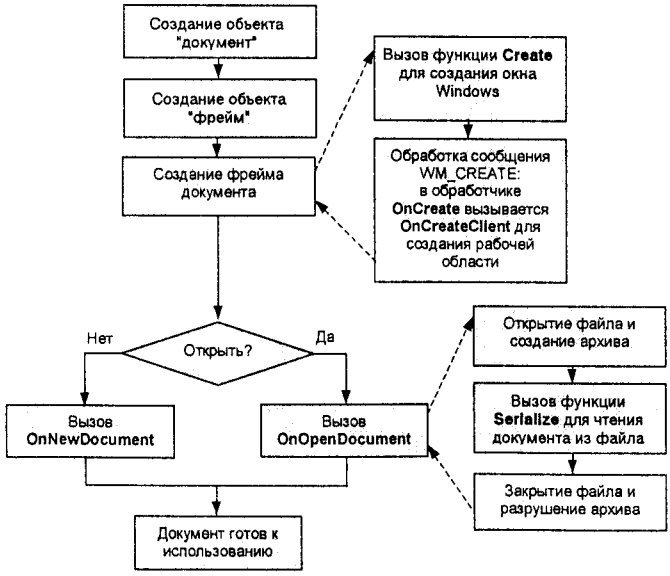


Рис. 37. Последовательность действий при создании документа

Данные документа определяются как переменные специального класса документа, производного от *CDocument*. Рассмотрим, как это реализовано в приложении NoteDraw, в котором используются документы двух типов. Для текстовых документов мы использовали класс библиотеки MFC *CTypedPtrList*, который представляет собой связанный список указателей на объекты произвольных типов:

```
class CNoteDoc : public CDocument
{
...
// данные текстовых документов
public:
// список указателей на объекты класса CParagraph,
// содержащего параметры форматирования абзаца текста
  CTypedPtrList<CObList, CParagraph*> m_listPar;
...
};
```

В данном случае *CTypedPtrList* содержит указатели на объекты специального созданного класса CParagraph, отвечающего за хранение и использование таких параметров как текст, его цвет, шрифт и стиль выравнивания для каждого абзаца текста.

Для графических документов используются другие типы данных:

```
class CDrawDoc : public CDocument
{
...
// данные графических документов
protected:
// параметры, используемые всеми нарисованными линиями
  int m_nWidth;
  CPen m_curPen;
  COLORREF m_curColor;
public:
// массив указателей на объекты класса CRect, содержащего
// координаты начальной и конечной точек рисуемой линии
  CArray<CRect*, CRect*> m_lineArray;
...
//
CPen  *GetPen() {return &m_curPen;}
...
};
```

Помимо самих данных, в классе документа зачастую определяются специальные функции для установки и извлечения элементов данных, а также для

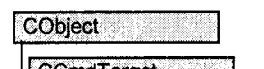
выполнения необходимых операций над ними (в приведенном фрагменте функция *GetPen)*.

К последнему действию, необходимому при создании какого-либо документа приложения — переопределению функции *Serialize* для реализации процесса сериализации — мы еще вернемся, а пока рассмотрим "цикл жизни" документа в рамках архитектуры "документ/представление":

- во время динамического создания вызывается конструктор объекта — "документ";
- для каждого нового документа вызывается функция *OnNewDocument* или *OnOpenDocument*;
- пользователь взаимодействует с документом посредством представлений, ассоциированных с ним;
- для удаления данных документа вызывается функция *DeleteContents*;
- для удаления объекта "документ" вызывается его деструктор.

Перечисленные шаги характерны для MDI-приложений. При работе с SDI-приложениями первый шаг выполняется единственный раз — при первом создании документа, а последний — когда приложение завершает свою работу.

Класс CDocument



Этот класс предоставляет базовые функциональные возможности для классов документов, определенных пользователем. Он поддерживает все стандартные операции, такие как создание документа, его загрузка и сохранение. Библиотека MFC работает с документами используя интерфейс, определенный в *CDocument.*

Объекты класса *CDocument* являются частью стандартного маршрута команд и поэтому получают команды от стандартных компонентов интерфейса пользователя. Документ получает команды раньше активного представления и если сам не обрабатывает некоторую команду, то передает ее на обработку своему шаблону.

Рассмотрим функции этого класса.

CDocument::**CDocument**() —
создает объект класса. Непосредственно обработку создания документа выполняет библиотека MFC. Для выполнения каких-либо специфических инициализирующих действий необходимо переопределить функцию OnNewDocument, что особенно важно для однодокументных приложений.

CDocTemplate* CDocument::**GetDocTemplate**() —
возвращает указатель на объект-шаблон для документа этого типа. Если документ этого типа не поддерживается каким-либо шаблоном, то возвращается NULL.

const CString& CDocument::**GetPathName**() —

возвращает полный путь к файлу документа. Строка будет пустой, если документ еще не был сохранен или с ним не ассоциирован никакой файл на диске.

virtual void CDocument::**SetPathName**(
LPCTSTR lpszPathName,
BOOL bAddToMRU = TRUE) —

устанавливает полный путь к файлу документа на диске (параметр lpszPathName). В зависимости от значения параметра bAddToMRU этот путь добавляется (TRUE) или не добавляется (FALSE) в список последних использованных файлов (MRU).

const CString& CDocument::**GetTitle**() —

возвращает название документа. Чаще всего оно образуется из имени соответствующего файла или берется из подстроки "Doc type name" соответствующей строки ресурса описания типа.

virtual void CDocument::**SetTitle**(LPCTSTR lpszTitle) —

устанавливает название документа (параметр lpszTitle). При этом происходит обновление заголовков всех фреймов, которые отображают документ.

BOOL CDocument::**IsModified**() —

возвращает TRUE, если документ был изменен после последнего сохранения, в противном случае — FALSE.

void CDocument::**SetModifiedFlag**(BOOL bModifitd = TRUE) —

позволяет пометить документ как модифицированный (bModified = TRUE), что позволит библиотеке MFC гарантировать вывод окна запроса пользователю на сохранение документа до его закрытия. Использование в качестве значения параметра FALSE позволяет пометить документ как немодифицированный, даже если в него были внесены какие-либо изменения, не отменяя их фактически.

void CDocument::**AddView**(CView *pView) —

присоединяет представление pView к документу, добавляя его к общему списку представлений, ассоциированных с этим документом. Помимо этого, функция устанавливает указатель m_pDocument класса CView на этот документ. Вызывается библиотекой MFC при создании нового представления документа, обычно обработчиками команд ID_FILE_NEW, ID_FILE_OPEN, ID_WINDOW_NEW или при разбиении разделяемого окна.

void CDocument::**RemoveView**(CView *pView) —

отсоединяет представление pView от документа и удаляет его от общего списка представлений, ассоциированных с этим документом. Указатель m_pDocument класса CView устанавливается в NULL. Вызывается библиотекой MFC при закрытии фрейма или области разделенного окна.

virtual POSITION CDocument::**GetFirstViewPosition**() —

позволяет получить позицию первого представления в общем списке представлений, ассоциированных с документом. Используется для возможности начала поиска требуемого представления.

virtual CView* CDocument::**GetNextView**(POSITION &rPosition) —

возвращает указатель на объект класса CView (или производного от него), определяемый его позицией rPosition в списке представлений документа. После этого устанавливает rPosition в значение для следующего представления в спи-

ске. Если полученное представление последнее в списке, то rPosition устанавливается в NULL. Позиция представления в списке полностью определяется порядком его записи в него.

```
...
// получаем указатель на документ, с которым
// ассоциировано текущее представление
  CNoteDoc* pDoc = (CNoteDoc*)GetDocument();
// получаем позицию первого представления, хранящегося в
// списке представлений ассоциированных с документом
  POSITION pos = pDoc->GetFirstViewPosition();
// получаем указатель на первое представление в списке
  pDoc->GetNextView(pos);
// получаем указатель на второе представление в списке
  CTextView *pText = (CTextView *)pDoc->GetNextView(pos);
...
// читаем данные представления
  pText->GetSelectedText(str);
...
```

void CDocument::**UpdateAllViews**(
CView *pSender,
LPARAM lHint = 0L,
CObject *pHint = NULL) —

информирует каждое представление, присоединенное к документу, за исключением модифицировавшего документ, на которое указывает pSender о том, что документ был изменен. Это приводит к вызову функции CView::OnUpdate для каждого представления (за исключением определяемого параметром pSender), передавая им информацию об изменениях в документе. Эта информация может быть представлена каким-либо кодом (параметр lHint) и/или содержаться в объекте (параметр pHint).

```
void CNoteView::OnPageSetup()
{
...
// после того как были изменены параметры страницы, необходимо
// произвести обновление изображения данных на экране
  CNoteDoc* pDoc = (CNoteDoc*)GetDocument();
// сообщение об обновлении передается всем представлениям,
// ассоциированным с документом, но реально предназначается
// только объекту класса, на который указывает this
  pDoc->UpdateAllViews(NULL, 0, this);
...
}
```

```
void CNoteView::OnUpdate(CView*, LPARAM, CObject *pHint)
{
 // если определено представление, требующее перерисовки
  if(pHint != NULL)

    // и оно является объектом класса CNoteView ...
    if(pHint->IsKindOf(RUNTIME_CLASS(CNoteView)))
    {
      // ... изменяем размеры этого представления
      ...
     }

  ...
}
```

virtual BOOL CDocument::**CanCloseFrame**(CFrameWnd *pFrame) —
вызывается библиотекой MFC до закрытия фрейма документа. Функция проверяет все фреймы, которые отображают документ, и если в нем были какие-либо изменения, выводит на экран окно запроса на сохранение. Параметр pFrame указывает на фрейм представления, присоединенного к документу.

virtual void CDocument::**DeleteContents**() —
удаляет данные документа без разрушения самого объекта. Вызывается библиотекой MFC непосредственно перед разрушением документа, а также гарантирует, что документ пустой, если есть необходимость в его многократном использовании. Для выполнения "осмысленных" действий необходимо переопределить эту функцию, т. к. реализация по умолчанию не делает ничего.

```
void CNoteDoc::DeleteContents()
{
// находим позицию первого абзаца в списке
  POSITION pos = m_listPar.GetHeadPosition();
  while(pos != NULL)
  {
  // последовательно проходим весь список
    CParagraph *pPar = m_listPar.GetNext(pos);

  // освобождаем память, занимаемую текущим абзацем;
  // если этого не сделать, то для системы память
  // будет потеряна, хотя Windows 95 и Windows NT
  // в большинстве случаев "уберут" за вами
    delete pPar;
  }

// очищаем список - делаем его "пустым"
  m_listPar.RemoveAll();

  // вызываем функцию базового класса для корректного
  // завершения
  CDocument::DeleteContents();
}
```

virtual void CDocument::**OnChangedViewList**() —
вызывается библиотекой MFC после того, как представление было добавлено в документ или удалено из него. В том случае, если текущее представление было последним в списке, удаляется весь документ.

virtual void CDocument::**OnCloseDocument**() —
вызывается библиотекой MFC, когда документ закрывается. Реализация по умолчанию вызывает функцию DeleteContents для удаления данных документа и закрывает все фреймы представлений, которые к нему присоединены.

virtual BOOL CDocument::**OnNewDocument**() —
вызывается библиотекой MFC при обработке команды ID_FILE_NEW. Реализация по умолчанию вызывает функцию DeleteContents, которая обеспечивает очистку документа, и соответствующим образом помечает его. При переопределении здесь рекомендуется инициализировать структуру данных нового документа. Для SDI-приложений эта функция вместо создания нового переинициализирует существующий объект документа. Для MDI-приложений библиотека MFC каждый раз создает новый объект документа, и в этой функции производится его инициализация.

```
BOOL CNoteDoc::OnNewDocument()
{
// всю стандартную обработку по созданию документа
// возлагаем на библиотеку MFC — она делает это прекрасно!
  if (!CDocument::OnNewDocument())
    return FALSE;

// после успешного создания проводим "настройку"
// конкретного документа
  InitDoc();

  return TRUE;
}
```

virtual BOOL CDocument::**OnOpenDocument**(LPCTSTR lpszPathName) —
вызывается библиотекой MFC при обработке команды ID_FILE_OPEN. Реализация по умолчанию открывает файл, имя которого определяется параметром lpszPathName, вызывает функцию DeleteContents, обеспечивающую очистку документа, функцию Serialize для чтения содержимого файла и помечает документ как немодифицированный. Переопределять ее следует, если используется какой-либо специальный механизм работы с архивами и/или файлами, например, распознавание формата документа.

virtual BOOL CDocument::**OnSaveDocument**(LPCTSTR lpszPathName) —
вызывается библиотекой MFC при обработке команд ID_FILE_SAVE или ID_FILE_SAVE_AS. Реализация по умолчанию открывает файл, имя которого определяется параметром lpszPathName, вызывает функцию Serialize для записи в файл данных документа и помечает его как немодифицированный.

Следующая группа функций достаточно редко используется в явном виде, и мы приводим их для полноты картины.

virtual void CDocument::**ReportSaveLoadException**(
LPCTSTR lpszPathName,
CException *e,
BOOL bSaving,
UINT nIDPDefault) —

вызывается библиотекой MFC, если во время выполнения операций сохранения или загрузки документа произошло исключение (CFileException или CArchiveException). Реализация по умолчанию проверяет объект исключения и ожидает от объекта исключения сообщения об ошибке. Если сообщение не найдено или параметр e, определяющий объект "исключение", равен NULL, то используется сообщение, задаваемое параметром nIDPDefault. Характер выполняемой операции задается параметром bSaving: сохранение — TRUE, загрузка — FALSE.

virtual CFile* CDocument::**GetFile**(
LPCTSTR lpszFileName,
UINT nOpenFlags,
CFileException pError) —

позволяет получить указатель на объект CFile по абсолютному или относительному пути lpszPathName к требуемому файлу. Параметр nOpenFlags задает режим доступа и совместного использования, а pError указывает на объект "исключение" при работе с файлом, в который записывается состояние завершения операции.

virtual void CDocument::**ReleaseFile**(
CFile *pFile,
BOOL bAbort) —

освобождает файл pFile, делая его доступным для других приложений. Параметр bAbort определяет функцию, при помощи которой будет освобождаться файл: FALSE — если используется функция CFile::Close, TRUE — при вызове функции CFile::Abort.

virtual BOOL CDocument::**SaveModified**() —

вызывается библиотекой MFC до того, как модифицированный документ будет закрыт. Реализация по умолчанию выводит на экран окно с запросом к пользователю на подтверждение сохранения изменений в документе. Этот запрос не выводится, если изменения в документе были блокированы вызовом функции SetModifiedFlag(FALSE).

virtual void CDocument::**PreCloseFrame**(CFrameWnd *pFrame) —

вызывается библиотекой MFC до разрушения фрейма pFrame, который владеет ассоциированным с ним документом. При переопределении этой функции желательно вызвать также соответствующую функцию базового класса.

Еще раз внимательно посмотрите на рис. 37. На нем наглядно показано, что документ может быть создан в двух случаях. Прежде всего, по команде ID_FILE_NEW создается новый, пустой документ. В этом случае для его инициализации достаточно переопределить функцию класса документа *OnNewDocument*. Во втором случае по команде ID_FILE_OPEN создается новый документ, содержимое которого загружается из файла. При этом для его инициализации можно переопределить функцию класса документа

OnOpenDocument. Этот случай представляет больший интерес, т. к. вплотную подводит нас к достаточно важному понятию не только в рамках архитектуры "документ/представление", но и во всей библиотеке классов MFC. Речь конечно же идет о сериализации, или, другими словами, преобразовании в последовательную форму. Мы уже рассматривали этот процесс в первом томе, поэтому напомним лишь ключевые моменты.[1] Основная идея сериализации заключается в том, чтобы обеспечить сохранение и восстановление текущего состояния объектов на устройстве постоянного хранения, например, в файле на диске. Поскольку речь идет о "преобразовании в последовательную форму", то очевидно, что состояние объекта сохраняется в бинарном формате. Следующим важным моментом, на котором мы остановимся, является то, что необходимо включить специальные макросы в объявление:

```
class CParagraph : public CObject
{
protected:
  CParagraph();

// Обязательный макрос, обеспечивающий поддержку сериализации
  DECLARE_SERIAL(CParagraph);

...
};
```

и реализацию класса:

```
// Обязательный макрос, обеспечивающий поддержку сериализации
IMPLEMENT_SERIAL(CParagraph, CObject, 2)
```

И, наконец, необходимо переопределить функцию *Serialize*, которая имеется у объекта любого класса, производного от *CObject*:

```
void CNoteDoc::Serialize(CArchive& ar)
{
  CNDApp *pApp = (CNDApp*)AfxGetApp();
  if (ar.IsStoring())
  {

// сохраняем текущий размер бумаги
    ar << pApp->m_sizePaper;
  }
  else
  {

// восстанавливаем текущий размер бумаги
    ar >> pApp->m_sizePaper;
  }
```

[1] Рекомендуем обратиться к разделу "Сериализация" главы 7.

```
// для сохранения и восстановления собственно содержимого
// документа вызываем функцию Serialize класса CObList,
// который, в свою очередь, последовательно вызывает
// аналогичную функцию для каждого абзаца.
// Последнюю тоже необходимо было переопределить,
// чтобы сохранить данные конкретного документа
  m_listPar.Serialize(ar);
}

void CParagraph::Serialize(CArchive& ar)
{
  LOGFONT lf;
  if (ar.IsStoring())
  {
    m_font.GetLogFont(&lf);

    ar << m_align;
    ar << m_color;
    ar << lf.lfCharSet;
    ar << lf.lfItalic;
    ar << lf.lfUnderline;
    ar << lf.lfStrikeOut;
    ar << lf.lfWeight;
    ar << lf.lfFaceName;
    ar << (long)(lf.lfHeight/2.3);
    ar << m_words;
  }
  else
  {
    CString tmp;
    memset(&lf, 0, sizeof(LOGFONT));
    ar >> m_align;
    ar >> m_color;
    ar >> lf.lfCharSet;
    ar >> lf.lfItalic;
    ar >> lf.lfUnderline;
    ar >> lf.lfStrikeOut;
    ar >> lf.lfWeight;
    ar >> tmp;
    strcpy(lf.lfFaceName, tmp);
    ar >> lf.lfHeight;
    lf.lfHeight = (long)(lf.lfHeight*17.55);
    ar >> m_words;

    m_font.CreatePointFontIndirect(&lf);
  }
}
```

Последний представленный фрагмент приведен нами для иллюстрации следующего положения. При сохранении параметров выбранного шрифта, которые хранятся в объекте класса *CFont*, было бы логично ожидать, что при использовании операторов >> и << архива, произошло бы сохранение (и восстановление) объекта *m_font*, поскольку класс *CFont* является производным от *CObject* и, следовательно, поддерживает сериализацию. Однако этого не происходит, т. к. в самом этом классе не переопределена функция *Serialize*, а аналогичные функции базовых классов, естественно, корректно не работают. Поэтому, прежде чем использовать сериализацию классов, производных от *CObject*, проверьте, реализована ли в них эта, такая удобная, а во многих случаях и необходимая, функция.

Теперь, для того чтобы завершить знакомство с документами, осталось рассмотреть класс *CArchive*. И хотя библиотека MFC уже создала его для нас к тому моменту, когда происходит вызов функции *Serialize*, может оказаться полезным создавать и использовать объекты этого класса самостоятельно.[1]

Класс CArchive

CArchive

Этот класс позволяет сохранять совокупную схему объектов в непрерывной бинарной форме — обычно в файле на диске. Архивный объект можно представлять как разновидность бинарного потока, и подобно потоку ввода/вывода он ассоциируется с файлом и дает возможность для буферизованного чтения/записи данных в/из хранилища. Но в отличие от потока ввода/вывода, обрабатывающего последовательность символов ASCII, архив обрабатывает данные бинарного объекта в эффективном безызбыточном формате.

До создания объекта *CArchive* необходимо создать объект класса *CFile* (для одного файла может использоваться только один активный архив). При этом необходимо также обеспечить, чтобы состояние загрузки/сохранения архива было совместимо с режимом открытия файла.

При конструировании объекта *CArchive* он присоединяется к объекту класса *CFile* или производного от него, в момент открытия файла. Кроме того, необходимо определить, будет ли архив использоваться для загрузки или сохранения. Объекты этого класса предназначены для работы не только с примитивными типами, но и с объектами классов, производных от *CObject*, подготовленными для сериализации, т. е. обычно имеющими функцию *Serialize* и использующими макросы DECLARE_SERIAL и IMPLEMENT_SERIAL.

Как обычно, рассмотрим основные члены этого класса.

CDocument CArchive::**m_pDocument** —
по умолчанию устанавливается в NULL, но по желанию пользователя может указывать на любой документ. Его основное назначение заключается в пере-

[1] Этот раздел имеет чисто справочный характер и может быть исключен при последовательном чтении.

даче дополнительной информации в процессе сериализации. Библиотека MFC устанавливает этот член в значение указателя на обрабатываемый (сериализуемый) документ, когда пользователь вызывает команду ID_FILE_OPEN или ID_FILE_SAVE.

CArchive::**CArchive**(
CFile *pFile,
UINT nMode,
int nBufSize = 4096,
void *lpBuf = NULL) —

конструктор создает объект класса и определяет, будет ли он использоваться для загрузки или сохранения объектов. Это назначение архива не может быть изменено после того, как он был создан. Более того, нельзя изменять также и состояние файла, пока архив не будет закрыт, т. к. любые операции такого рода портят целостность архива. В качестве параметров в конструктор передаются: pFile — указатель на объект CFile, который является окончательным приемником или источником сохраняемых данных; nBufSize — целое число, определяющее размер (в байтах) внутреннего буфера файла (по умолчанию имеет размер 4096 байт); lpBuf — необязательный указатель на буфер размера nBufSize, если использовать значение по умолчанию (NULL), то архив сам запросит под него требуемый объем памяти и сам же освободит ее при разрушении объекта; nMode — флаг, который определяет, будет ли объект загружаться из архива или сохраняться в нем, должен иметь одно из следующих значений:

CArchive::**load**	загружает данные из архива; требует только чтения CFile
CArchive::**store**	сохраняет данные в архиве; требует записи CFile
CArchive::**bNoFlushOnDelete**	блокирует архив от автоматического вызова функции Flush при вызове его деструктора; если этот флаг установлен, то до вызова деструктора необходим явный вызов функции Close, в противном случае данные могут быть испорчены

void CArchive::**Abort**() —

закрывает архив без обработки исключения. Деструктор объекта вызывает функцию Close, которая принудительно записывает в файл любые несохраненные данные, ассоциированные с объектом CFile, что может вызвать исключение. Если архив создавался динамически (с использованием оператора new), то после закрытия файла необходимо освободить память.

void CArchive::**Close**() —

принудительно записывает в файл любые данные, оставшиеся в буфере, закрывает архив и отсоединяет его от файла. После закрытия архива можно создать другой для того же самого файла или закрыть файл. Выполнение этой функции гарантирует, что все данные переданы из архива в файл, и сам архив больше не доступен. Для сохранения данных на устройстве постоянного хранения (чаще всего на диске) сначала необходимо использовать функцию CFile::Close и только после этого разрушить объект с помощью функции CFile.

void CArchive::**Flush**() —

принудительно записывает в файл все данные, оставшиеся в буфере архива. Для полной гарантии записи файла на диск (устройство постоянного хранения) необходимо вызвать функцию CFile::Close.

В классе реализованы несколько вариантов перегруженных операторов >> и <<, которые, соответственно, загружают указанный объект (или примитивный тип) из архива, или сохраняют его в нем. Они имеют достаточно однотипные формы, некоторые из которых представлены ниже:

```
friend CArchive& operator <<(
  CArchive &ar,
  Cobject *&pOb);

friend CArchive& operator >>(
  CArchive &ar,
  Cobject *&pOb);

CArchive& operator << (Data_Type& data);

CArchive& operator >> (Data_Type& data);
```

Через Data_Type здесь обозначены примитивные типы данных BYTE, WORD, int, DWORD, float и double.

Для работы с более сложными структурами данных в классе реализованы функции

```
UINT CArchive::Read(
  void *lpBuf,
  UINT nMax) —
```

читает из архива в буфер (параметр lpBuf) определенное число (параметр nMax) байт. Функция возвращает действительно прочитанное число байт, которое может быть меньше заданного при достижении конца файла. Ее можно использовать внутри переопределенной функции Serialize для чтения простых структур, содержащихся в объекте.

```
void CArchive::Write(
  const void *lpBuf,
  UINT nMax) —
```

записывает из буфера (параметр lpBuf) в архив определенное число (параметр nMax) байт. Ее можно использовать внутри переопределенной функции Serialize для записи простых структур, которые содержатся в объекте.

Поскольку строки являются одними из наиболее часто используемых типов данных, то для работы с ними реализованы специальные функции:

```
BOOL CArchive::ReadString(CString &rString)
```

и

```
LPTSTR CArchive::ReadString(
  LPTSTR lpsz,
  UINT nMax) —
```

читают текстовые данные в буфер из файла, ассоциированного с объектом CArchive. Чтение прекращается, если прочитана пара символов "возврат карет-

ки — перевод строки", которые в буфер не записываются. Вместо них к прочитанной строке добавляется "\0". Для второго варианта функции буфер должен иметь размер не менее (nMax — 1) символов.

void CArchive::**WriteString**(LPTSTR lpsz) —
записывает данные из буфера, который содержит ограниченную нулем строку, в файл, ассоциированный с объектом CArchive. При этом вместо нулевого символа в файл записывается признак конца строки — "\n".

Так как архив может быть открыт либо для чтения, либо для записи данных (но не одновременно), то необходимо иметь информацию о его "предназначении", что обеспечивается двумя функциями

BOOL CArchive::**IsLoading**() —
определяет, был ли архив открыт для загрузки данных. Она вызывается из функций Serialize классов, поддерживающих сериализацию.

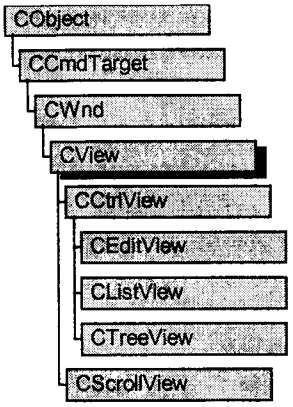
BOOL CArchive::**IsStoring**() —
определяет, был ли архив открыт для сохранения данных. Она вызывается из функций Serialize классов, поддерживающих сериализацию.

```
void CDrawDoc::Serialize(CArchive& ar)
{
  // действия, относящиеся и к загрузке и к сохранению архива
  ...
  if (ar.IsStoring())
  {
    // сохранение данных в архиве
    ...
  }
  else
  {
    // загрузка данных из архива
    ...
  }
}
```

Закончив знакомство с документами, пора переходить к тому, каким образом можно взаимодействовать с хранящимися в них данными. А осуществить это взаимодействие и визуализацию данных документа призваны объекты специальных классов — представления.

Представления

Это специальная группа классов дочерних окон фрейма, которая отвечает за отображение данных документа и за взаимодействие с пользователем. В этой главе мы представим те возможности, доступные для широкого применения, которыми располагает библиотека классов MFC.



Представление ассоциируется с некоторым документом и действует как посредник между ним и пользователем. Другими словами, представление переводит образ документа на экран, принтер или любое другое устройство графического вывода, и интерпретирует действия пользователя как операции над документом. Основным, на что здесь следует обратить внимание, является то, что представление может быть *ассоциировано* (или *присоединено*) только с *одним* документом, т. е. хотя оно и является окном, как, впрочем, почти все видимые объекты Windows, но при этом ориентировано на работу только с определенным набором данных. С другой стороны, в рамках одного фрейма документа можно создавать и использовать сколько угодно представлений для работы с одним и тем же документом.

Достаточно часто возникает желание иметь различные типы представлений одного, данного, типа документа. Что имеется в виду? Например, при работе с текстовым процессором желательно иметь одно представление для текста документа, а другое (контурное) — для изображения области колонтитулов. Эти различные типы представлений могут размещаться как в различных фреймах, так и в различных областях одного и того же фрейма. В обоих случаях достаточно один раз сопоставить представление с документом, но организовать различную обработку данных документа (рис. 38).

Таким образом, представление отвечает за изображение и модификацию данных документа, но не имеет никакого отношения к их хранению.[1] Необходимые подробности о различных параметрах этих данных хранятся вместе с самим документом, откуда либо само представление, либо специальные функции класса документа, обеспечивающие для того или иного представления доступ к ним, берут только необходимое.

Несколько слов о взаимодействии с пользователем. Реализация выполнения любых действий такого рода целиком ложится на соответствующий класс представления, а, значит, и на программиста.[2] Это может быть ввод с клавиатуры, действия мышью, выбор элемента меню, кнопки панели инструментов или работа с полосой прокрутки. Соответствующие команды представление получает от своего фрейма (правда не напрямую, а опосредованно). Обработку сообщений оно осуществляет аналогично всем другим получателям команд — посредством карты сообщений, что подробно описано нами в первом томе. Если представление само не обрабатывает команду, то передает ее дальше "по маршруту", как описано в главе 10.

---

[1] Точнее, не должно иметь.

[2] Никто не может в полной мере предугадать всех ваших желаний, даже разработчики прекрасной библиотеки классов MFC.

Перед знакомством с классами представлений (и связанных с ними), приведем последовательность действий по их созданию, выполняемую библиотекой классов MFC (рис. 38).

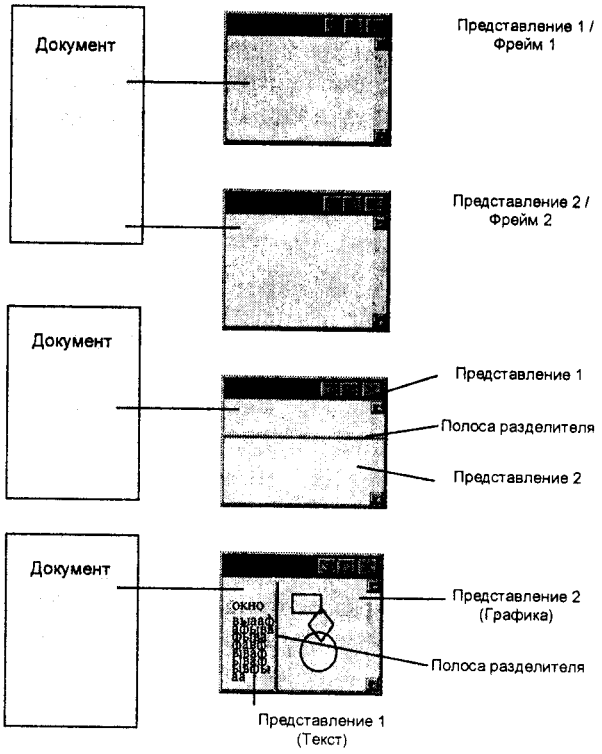


Рис. 38. Использование нескольких представлений для отображения данных одного документа

Теперь, после того как сложилось некоторое общее представление о назначении и возможностях этой группы классов, можно переходить к рассмотрению состава этой группы и свойств ее составляющих.

В полном соответствии с общей идеологией построения библиотеки MFC на вершине классов представлений стоит единственный базовый класс — *CView*, обеспечивающий поддержку печати, общее взаимодействие с документом и некоторые другие возможности, которые будут описаны ниже. От него образованы еще два класса — *CCtrlView* и *CScrollView*, каждый из которых унаследовал все свойства базового класса и добавил свои специфические. Класс *CCtrlView* добавлен в библиотеку классов MFC версии 4.0 для того, чтобы можно было использовать в архитектуре "документ/представление" деревья (класс *CTreeView*), списки (*CListView*), а также простейший (*CEditView*) и расширенный (*CRichEditView*) элементы управления для редактирования текстов. Класс *CScrollView* добавляет к свойствам базового класса

поддержку автоматической прокрутки документов, если данные документа, которые необходимо отобразить, не помещаются в одном окне. Производный от него класс — *CFormView* обеспечивает дополнительную поддержку диалоговых ресурсов, таких как кнопки и поля. Наибольшее применение этот класс находит при работе с базами данных, о чем свидетельствуют его производные классы: *CDaoRecordView* и *CRecordView*. Поскольку мы исключили работу с базами данных из вопросов, обсуждаемых в этой книге, то не будем останавливаться на рассмотрении этих классов.

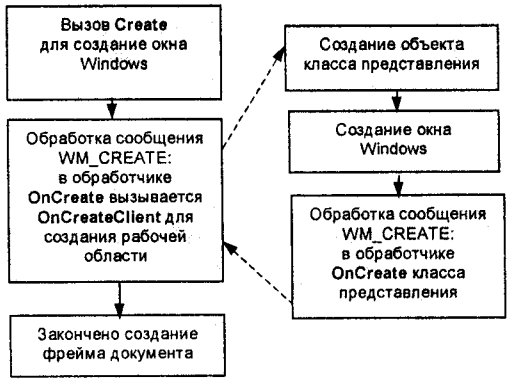


Рис. 39. Последовательность создания представления

Таким образом, в библиотеке классов MFC реализованы десять классов представлений, из которых мы подробно рассмотрим семь (класс *CRichEditView* будет описан в третьем томе). Кроме того, здесь же будет рассмотрен класс *CSplitterWnd*, который хотя и не входит в группу классов представлений, но достаточно тесно с ними связан.

Класс CView

Этот класс предоставляет базовые функциональные возможности для всех классов представлений, которые есть в библиотеке или определяются пользователем. Рассмотрим основные функции этого класса.

Первым, как всегда, идет конструктор.

CView::**CView**() —
: создает объект класса; вызывается, когда создается новый фрейм или окно разделяется на области. Никакой инициализации представления здесь не производится.

CDocument* CView::**GetDocument**() —
: позволяет получить указатель на объект "документ", присоединенный к этому представлению. Если с ним не ассоциирован никакой документ, то возвращает-

ся NULL. Использование этого указателя предоставляет доступ к функциям соответствующего класса документа. Для каждого производного класса создается своя специфичная функция.

virtual void CView::**OnInitialUpdate**() —

вызывается библиотекой MFC после того, как представление первый раз присоединено к документу, но до его первоначального отображения. Реализация по умолчанию вызывает функцию CView::OnUpdate без какой-либо информации (lHint = 0 и pHint = NULL). Для проведения специальных инициализирующих действий необходимо переопределить эту функцию. Например, так:

```
void CNoteView::OnInitialUpdate()
{
// Пусть библиотека MFC выполнит за нас всю
// необходимую предварительную работу
  CScrollView::OnInitialUpdate();

// Создаем новый (пустой) абзац
  CNoteDoc* pDoc = (CNoteDoc*)GetDocument();
  CParagraph *pPar = pDoc->OnNewPar(ND_LEFT);

// Добавляем его в общий список абзацев
  pDoc->m_listPar.AddTail(pPar);

// Обновляем все изображения данных документа
  pDoc->UpdateAllViews(NULL, 0, this);
}
```

virtual void CView::**OnUpdate**(
CView *pSender,
LPARAM lHint,
CObject *pHint) —

вызывается из CDocument::UpdateAllViews после того, как документ был модифицирован. Кроме того, она вызывается из функции CView::OnInitialUpdate. Реализация по умолчанию помечает всю рабочую область как недействительную для ее перерисовки при получении следующего сообщения WM_PAINT. Если обновления требует не вся рабочая область, то следует переопределить эту функцию, посылая информацию об изменениях через параметры lHint и pHint. Параметр lHint определяет специальное значение, обычно битовую маску или перечислимый тип, несущее информацию о характере изменений в документе. Параметр pHint определяет указатель на объект, производный от CObject, который хранит информацию об изменениях в документе. При переопределении этой функции можно воспользоваться функцией CObject::IsKindOf для получения типа времени выполнения этого объекта. Если оба эти параметра нулевые, то документ послал общее извещение об обновлении. Представление, получившее такое извещение или не сумевшее декодировать параметры, обновляет всю свою рабочую область. Параметр pSender идентифицирует представление, которое должно быть обновлено в соответствии с изменениями в документе.[1]

---

[1] Обратите внимание! Параметр *pSender* рассмотренной ранее функции *CDocument::UpdateAllViews* исключает представление, на которое указывает, из группы представлений, получающих сообщение об обновлении, а параметр *pSender* функции *CView::OnUpdate*, наоборот, идентифицирует представление, получившее это сообщение.

```
void CNoteView::OnUpdate(CView*, LPARAM, CObject *pHint)
{
  if(pHint != NULL)
  {
    // нам ли предназначено сообщение?
    if(pHint->IsKindOf(RUNTIME_CLASS(CNoteView)))
    {
      // здесь можно проводить любые необходимые действия,
      // необходимые для инициализации представления CNoteView
      ...
    }
  }
  // Обновляем все, хотя можно и ограничить размеры области
  Invalidate(TRUE);
}
```

В этой функции можно выполнять непосредственно и "перерисовку" данных документа. Однако для этих целей предназначены другие функции. Здесь лучше просто описать (в координатах устройства) прямоугольную область, требующую перерисовки, и передать ее в функцию InvalidateRect, чтобы осуществить рисование при получении следующего сообщения WM_PAINT.

virtual void CView::**OnActivateView**(
BOOL bActivate,
CView *pActivateView,
CView *pDeactiveView) —

вызывается библиотекой MFC, когда представление активизируется (bActivate=TRUE) или деактивизируется. По умолчанию функция устанавливает фокус на активизируемое представление. В качестве параметров она получает также указатели на активизируемое (параметр pActivateView) и деактивизируемое (параметр pDeactiveView) представления. Эти параметры указывают на одно и то же представление, если активизируется главное окно приложения без изменения активного представления, что можно использовать для изменения свойств представления. Но эти параметры отличаются при переключении между различными представлениями одного и того же приложения (или дочерними окнами MDI-приложения). Наиболее часто это связано с "разделяемыми" (splitter) окнами.

virtual void CView::**OnActivateFrame**(
UINT nState,
CFrameWnd *pFrameWnd) —

вызывается в случае, когда активизируется или деактивизируется фрейм (параметр pFrameWnd), ассоциированный с текущим представлением. Характер действия определяется параметром nState, который может принимать одно из значений:

WA_INACTIVE	фрейм деактивизируется
WA_ACTIVE	фрейм активизируется любым способом, кроме как от нажатия в нем кнопки мыши, например, от клавиатуры

WA_CLICKACTIVE	фрейм активизируется путем нажатия кнопки мыши в его границах

virtual BOOL CView::**OnScroll**(
UINT nScrollCode,
UINT nPos,
BOOL bDoScroll) —

вызывается библиотекой классов MFC для определения возможности прокрутки в двух случаях, в зависимости от значения параметра bDoScroll, которое определяет, нужно ли действительно осуществлять прокрутку. В первом случае (bDoScroll = TRUE), когда представление получает сообщение о прокрутке, действительно нужно осуществить прокрутку представления. Во втором случае (bDoScroll = FALSE), когда элемент OLE переносится в область автопрокрутки, осуществлять саму прокрутку представления не нужно. Параметр nScrollCode определяет код прокрутки и состоит из двух частей: младший байт задает код прокрутки по горизонтали, а старший — по вертикали. Он может принимать одно из следующих значений:

SB_BOTTOM	прокрутка до самого низа
SB_TOP	прокрутка до самого верха
SB_LINEDOWN	прокрутка вниз на одну линию (строку)
SB_LINEUP	прокрутка вверх на одну линию (строку)
SB_PAGEDOWN	прокрутка вниз на одну страницу
SB_PAGEUP	прокрутка вверх на одну страницу
SB_THUMBTRACK	перенос бегунка полосы прокрутки в определенную позицию, указанную в параметре *nPos*

Функция возвращает TRUE, если прокрутка действительно осуществлена; в противном случае — FALSE.

virtual BOOL CView::**OnScrollBy**(
CSize sizeScroll,
BOOL bDoScroll) —

вызывается библиотекой классов MFC, когда курсор мыши находится вне области, изображаемой представлением документа, что имеет место или при переносе элемента OLE к границе текущего представления, или при манипуляциях горизонтальной или вертикальной полосами прокрутки. Реализация по умолчанию не делает ничего. В производных классах функция проверяет, видимо ли представление, прокручиваемое в направлении, запрошенном пользователем, и, при необходимости, обновляет новый регион. Она автоматически вызывается функциями CWnd::OnHScroll и CWnd::OnVScroll для выполнения прокрутки. Параметр sizeScroll определяет число пикселей (по горизонтали и по вертикали) на которое осуществляется прокрутка. Функция возвращает TRUE, если представление может быть прокручено, и FALSE в противном случае.

virtual void CView::**OnDraw**(CDC *pDC) = 0 —

чисто виртуальная функция, которая используется для изображения образа документа. Следует *обязательно переопределить* ее для изображения представления документа, что и сделано в производных классах. Библиотека MFC использует эту функцию как для печати (и предварительного просмотра)

документа, так и для отображения его на экране. Это основная и единственная функция для изображения видимого образа документа. От того, каким образом она реализована, зависит, что увидит пользователь на экране или после печати на принтере.

```
void CDrawView::OnDraw(CDC* pDC)
{
  CRect *pLine;

// Поскольку отобразить нужно данные документа,
// то получаем указатель на него
  CDrawDoc *pDoc = (CDrawDoc*)GetDocument();

// Мы могли нарисовать несколько объектов,
// и все их нужно вывести на экран, поэтому организуем цикл
  for(int i = 0; i< pDoc->m_lineArray.GetSize(); i++)
  {
    // получаем координаты текущей линии
    pLine = pDoc->m_lineArray[i];

    // выбираем в контекст устройства, которым может быть
    // как экран, так и принтер, текущий карандаш
    CPen *pOld = pDC->SelectObject(pDoc->GetPen());

    // выводим текущую линию на экран
    pDC->MoveTo(pLine->left,  pLine->top);
    pDC->LineTo(pLine->right, pLine->bottom);

    // восстанавливаем параметры "старого" карандаша,
    pDC->SelectObject(pOld);
  }
}
```

Внимательно ознакомившись с представленными функциями, нетрудно заметить, что очень многие из них должны быть переопределены в производных классах, что и сделано в оставшихся классах группы представлений.

Класс CCtrlView

Это базовый класс для классов элементов управления, включающих *CEditView*, *CListView*, *CRichEditView* и *CTreeView*, которые специально адаптированы к архитектуре "документ/представление". Его назначение во многом напоминает назначение класса *CControlBar* для панелей элементов управления, который подробно был рассмотрен в главе 13. Для самого класса *CCtrlView* приведем две общедоступные переменные и только конструктор:

CString CCtrlView::**m_strClass** —
содержит имя класса окна Windows для класса представления.

DWORD CCtrlView::**m_dwDefaultStyle** —
содержит стиль, устанавливаемый по умолчанию для класса представления. Этот стиль применяется при создании соответствующего окна.

CCtrlView::**CCtrlView**(
LPCTSTR lpszClass,
DWORD dwStyle) —

создает объект класса. Этот конструктор вызывается библиотекой MFC при создании нового фрейма или разбиении окна на области. Для инициализации представления, после того как оно присоединено к соответствующему документу, необходимо переопределить функцию CView::OnInitialUpdate, а для создания объекта Windows следует использовать функцию CWnd::Create или CWnd::CreateEx. Передаваемые в конструктор параметры определяют имя класса Windows — lpszClass и стиль, используемый по умолчанию, — dwStyle и записывают эти значения в соответствующие переменные класса.

Помимо перечисленных членов класса, в нем еще переопределены функции *OnDraw* и *PreCreateWindow*, а также реализован обработчик *OnPaint* сообщения WM_PAINT.

Как видите, этот класс служит лишь для установки общих параметров, а вся основная работа ложится на его производные классы, которые мы и рассмотрим.

Класс CEditView

Этот класс определяет представление, которое, подобно классу *CEdit*, инкапсулирует функциональные возможности элемента управления EDIT и может быть использовано для реализации простейшего текстового редактора. При этом, поскольку он создан на базе *CView*, объекты этого класса могут использоваться с документами и шаблонами документов, как и любое другое представление. Помимо тех возможностей, которые он унаследовал от своих "родительских" классов, в нем дополнительно реализована поддержка функций поиска и замены фрагментов текста и обработка стандартных команд ID_EDIT_SELECT_ALL, ID_EDIT_REPLACE, ID_EDIT_REPEAT, ID_FILE_PRINT и ID_EDIT_FIND.

Присущие элементу управления EDIT ограничения в полной мере относятся и к рассматриваемому классу. Объект класса *CEditView* (или производного от него) имеет следующие ограничения:

- не поддерживает редактирование по принципу "что вижу, то и получаю" (What You See Is What You Get, WYSIWYG), т. к. предназначен, в основном, для чтения с экрана;
- может изображать текст только одним шрифтом и не поддерживает специального форматирования символов;
- так же, как для элемента управления EDIT, размер текста, который может содержать объект класса *CEditView*, ограничен.

После небольшого обзора, в результате которого должно было сложиться общее представление о возможностях класса, рассмотрим его основные члены.

static AFX_DATA const DWORD CEditView::**dwStyleDefault** —

определяет стиль, используемый по умолчанию, для всех объектов класса CEditView.

CEditView::**CEditView**() —

конструктор, создает объект класса. После создания объекта, но до использования элемента управления EDIT необходимо вызвать функцию CWnd::Create для создания окна Windows. Если на базе этого класса определен производный, который добавлен в шаблон документа при помощи функции CWinApp::AddDocTemplate, то библиотека сама вызывает и конструктор и функцию Create.

Рассматриваемый класс до последнего времени (точнее, до появления 32-разрядных версий Windows) являлся одним из наиболее часто используемых, и поэтому достаточно полно адаптирован для различных режимов работы. В нем реализована поддержка печати, предварительного просмотра, установка величины табуляции и многое другое.[1]

CFont* CEditView::**GetPrinterFont**() —

возвращает указатель на объект CFont, который описывает текущий шрифт принтера, или NULL, если его шрифт не был установлен. В последнем случае печать осуществляется экранным шрифтом.

void CEditView::**SetPrinterFont**(CFont* pFont) —

устанавливает для принтера шрифт, определяемый указателем pFont на объект класса CFont. Ее следует вызывать из функции OnPreparePrinting для того, чтобы установка нового шрифта произошла до начала печати содержимого представления. Используется в случае, когда для принтера нужен особый шрифт. Если параметр pFont равен NULL, то шрифт, используемый для печати, основывается на экранном шрифте.

UINT CEditView::**PrintInsideRect**(
CDC *pDC,
RECT &rectLayout,
UINT nIndexStart,
UINT nIndexStop) —

печатает текст в прямоугольнике, заданном в параметре rectLayout. Если у объекта класса не установлен стиль ES_AUTOHSCROLL, то текст переносится на следующую строку внутри прямоугольника. В противном случае текст отсекается по его правой границе. Поле rectLayout.bottom изменяется так, что размеры определяют ту часть начального прямоугольника, которую занимает текст. Параметры nIndexStart и nIndexStop показывают, соответственно, индексы первого и следующего за последним полученным символов относительно начала буфера. Функция возвращает индекс следующего символа, который будет печататься.

void CEditView::**SetTabStops**(int nTabStops) —

устанавливает величину табуляции (параметр nTabStops) в единицах блока диалога (1/4 среднего значения ширины символов используемого шрифта), используемую и для экрана, и для принтера. Этот класс поддерживает только одно значение табуляции, в отличие от класса CEdit.

---

[1] Он же является ярким примером нарушения принципа разделения данных документа и их изображения.

void CEditView::**GetSelectedText**(CString &strResult) —

копирует выделенный текст или символы, предшествующие первому символу "возврат каретки", в объект класса CString, на который ссылается параметр strResult.

UINT CEditView::**GetBufferLength**() —

возвращает число символов (без завершающего нулевого), находящихся в текущий момент в буфере элемента управления EDIT.

LPCTSTR CEditView::**LockBuffer**() —

блокирует буфер элемента управления EDIT в определенном месте памяти, запрещая его модификацию и перемещение, и возвращает указатель на него.

void CEditView::**UnlockBuffer**() —

разрешает модификацию и перемещение буфера, заблокированного функцией LockBuffer.

void CEditView::**SerializeRaw**(CArchive &ar) —

служит для сериализации содержимого объекта CEditView в текстовый файл. Параметр ar хранит преобразованный текст. Эта функция является внутренней реализацией функции Serialize для класса CEditView и отличается от нее тем, что читает и записывает только текст без предварительного описания данных объекта. Таким образом, для этого представления данные хранятся именно в нем, а не в документе.

Еще одна группа функций облегчает работу с блоком диалога Find/Replace (Поиск/Замена), который выводится на экран в ответ на стандартные команды ID_EDIT_FIND и ID_EDIT_REPLACE, обеспечивая программистов обработчиками команд кнопок Find (Найти), Replace (Заменить) и Replace All (Заменить все).

BOOL CEditView::**FindText**(
LPCTSTR lpszFind,
BOOL bNext = TRUE,
BOOL bCase = TRUE) —

осуществляет поиск текста, заданного параметром lpszFind, в буфере элемента управления, начиная с текущей позиции и в направлении, определяемом параметром bNext; если он равен TRUE, то поиск осуществляется от начала к концу буфера, в противном случае — наоборот. Параметр bCase определяет, следует ли различать строчные и прописные буквы при поиске (TRUE — учитывать, FALSE — не учитывать регистр). Если в результате проведенного поиска текст не найден, то функция возвращает FALSE. Обычно она вызывается из переопределенной функции OnFindText.

virtual void CEditView::**OnFindText**(
LPCTSTR lpszFind,
BOOL bNext = TRUE,
BOOL bCase = TRUE) —

аналогична предыдущей функции, однако вызывается в ответ на нажатие кнопки Find Next (Следующий) в стандартном блоке диалога Find (Поиск), выводимого на экран в ответ на выбор команды ID_EDIT_FIND. По умолчанию просто вызывает функцию FindText. Если текст не найден, то вызывается функция OnTextNotFound.

virtual void CEditView::**OnReplaceAll**(
LPCTSTR lpszFind,
LPCTSTR lpszReplace,
BOOL bCase) —
вызывается, когда пользователь выбирает кнопку Replace All (Заменить все) в стандартном блоке диалога Replace (Замена). Она ищет в буфере текст, заданный параметром lpszFind, начиная с текущей позиции, и, при нахождении, заменяет его на другой, определенный в lpszReplace. Параметр bCase задает чувствительность к регистру. Поиск осуществляется посредством вызова функции FindText. В случае, если заданный текст не найден, вызывается функция OnTextNotFound. Для выполнения каких-либо специфических действий эту функцию следует переопределить.

virtual void CEditView::**OnReplaceSel**(
LPCTSTR lpszFind,
BOOL bNext,
BOOL bCase,
LPCTSTR lpszReplace) —
вызывается, когда пользователь выбирает кнопку Replace (Заменить) в стандартном блоке диалога Replace (Замена). После замены *выделенного* текста функция ищет следующий фрагмент, совпадающий с определяемым параметром lpszFind в направлении, определяемом параметром bNext. Сам поиск осуществляется посредством функции FindText. В случае, если заданный текст не найден, вызывается функция OnTextNotFound. Для выполнения каких-либо специфических действий эту функцию следует переопределить.

virtual void CEditView::**OnTextNotFound**(LPCTSTR lpszFind) —
вызывается библиотекой MFC, если текст, определенный в параметре lpszFind, не найден. По умолчанию вызывает функцию Windows MessageBeep. Если нужны какие-либо другие действия, функцию следует переопределить.

Работа с представлением, реализованном на базе класса *CEditView*, практически полностью совпадает с работой с элементом управления EDIT. Поэтому не будем останавливаться на этом классе более подробно, а в качестве примера работы с классом, рассмотрим фрагменты кода приложения NoteDraw, которые демонстрируют, каким образом осуществляется организация режимов поиска и замены в рассматриваемом представлении. На рис. 40 и 41 представлен внешний вид блоков диалога Find и Replace, соответственно в режимах поиска и замены.

Прежде всего необходимо зарегистрировать новое сообщение[1]

```
// для работы с блоком диалога Find или Replace
// при регистрации сообщения необходимо использовать
// строковую константу FINDMSGSTRING
static const UINT nNewMsg =
                ::RegisterWindowMessage(FINDMSGSTRING);
```

[1] Более подробную информацию можно получить в 15 главе, где рассматриваются стандартные блоки диалога.

В карту сообщений класса необходимо добавить макрокоманды, связывающие команды с соответствующими обработчиками:

```
BEGIN_MESSAGE_MAP(CTextView, CEditView)
  //{{AFX_MSG_MAP(CTextView)
  ...
  ON_COMMAND(ID_EDIT_FIND, OnEditFind)
  ON_COMMAND(ID_EDIT_REPLACE, OnEditReplace)
  //}}AFX_MSG_MAP
  ON_REGISTERED_MESSAGE(nNewMsg, OnNewCmd)
END_MESSAGE_MAP()
```

И, наконец, в реализацию класса необходимо добавить обработчики команд:

```
void CTextView::OnEditFind()
{
  // создаем объект "блок диалога"
  CFindReplaceDialog *pDlg = new CFindReplaceDialog;
  bFindOnly = TRUE;

  // создаем объект Windows "блок диалога" Find (Поиск)
  pDlg->Create(TRUE, NULL, NULL, FR_HIDEWHOLEWORD|FR_DOWN, this);
}
```

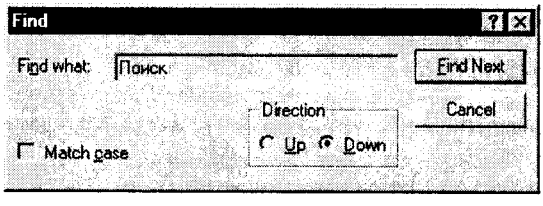


Рис. 40. Стандартный блок диалога Find

```
void CTextView::OnEditReplace()
{
  // создаем объект "блок диалога"
  CFindReplaceDialog *pDlg = new CFindReplaceDialog;
  bFindOnly = FALSE;

  // создаем объект Windows "блок диалога" Replace (Замена)
  pDlg->Create(FALSE, NULL, NULL, FR_NOWHOLEWORD | FR_DOWN, this);
}

LRESULT CTextView::OnNewCmd(WPARAM, LPARAM lParam)
{
// получаем указатель на объект "стандартный блок диалога"
// Find/Replace (Поиск/Замена)
  CFindReplaceDialog* pDialog =
                CFindReplaceDialog::GetNotifier(lParam);
```

```
  if (pDialog->IsTerminating())
  {
    // если пользователь завершил работу по поиску/замене, то
    // сообщаем библиотеке, что объект "блок диалога"
    // больше не нужен
    pDialog = NULL;
  }
  else if (pDialog->FindNext())
  {
    // получаем строку, которую требуется найти
    CString strFind = pDialog->GetFindString();

    // осуществляем поиск;
    // параметры поиска получаем прямо из блока диалога
    OnFindNext(strFind,
               pDialog->SearchDown(),
               pDialog->MatchCase());
  }
  else if (pDialog->ReplaceCurrent())
  {
    // сюда вход разрешен только для режима замены
    ASSERT(!bFindOnly);

    // пользователь нажал кнопку Replace (Заменить): требуется
    // найти определенную строку и заменить ее на заданную;
    // параметры получаем прямо из блока диалога
   OnReplaceSel(pDialog->GetFindString(),pDialog->SearchDown(),
                pDialog->MatchCase(),pDialog->GetReplaceString());
  }
  else if (pDialog->ReplaceAll())
  {
    ASSERT(!bFindOnly);

    // пользователь нажал кнопку Replace All (Заменить все):
    // требуется найти все вхождения определенной строки
    // и заменить их на заданную;
    // параметры получаем прямо из блока диалога
    OnReplaceAll(pDialog->GetFindString(),
                 pDialog->GetReplaceString(),
                 pDialog->MatchCase());
  }

  return 0;
}

void CTextView::OnTextNotFound(LPCTSTR lpszFind)
{
  // если искомая строка не найдена, то выводим об этом
  // информационное сообщение
  AfxMessageBox("Текст не найден");
}
```

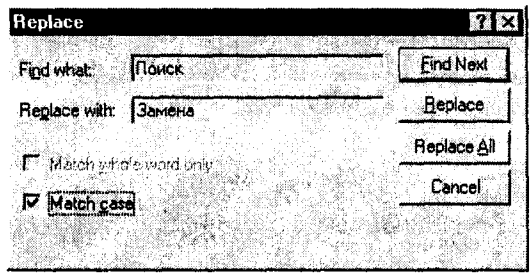


Рис. 41. Стандартный блок диалога Replace

Классы CListView и CTreeView

Эти классы позволяют значительно упростить использование элементов управления Windows — LIST VIEW и TREE VIEW и инкапсулировавших их функциональные возможности классов *CListCtrl* и *CTreeCtrl* в архитектуре "документ/представление".

Они содержат всего по две функции:

CListView::**CListView**()

и

CTreeView::**CTreeView**() —

создают объекты соответствующего класса.

CListCtrl& CListView::**GetListCtrl**()

и

CTreeCtrl& CTreeView::**GetTreeCtrl**() —

позволяют получить доступ к элементам управления LIST VIEW и TREE VIEW, ассоциированным с соответствующими представлениями.

Все остальные возможности они наследуют от своих базовых классов и соответствующих элементов управления.

Класс CScrollView

Согласитесь, что достаточно редко нам "хватает" одного окна для отображения всей необходимой информации. Гораздо чаще мы вынуждены организовать либо прокрутку документа, либо изменение его масштаба, либо чередовать и то и другое. Если базироваться на классе *CView*, то задача такого рода целиком ложится на плечи программиста: необходимо самому обрабатывать сообщения от полос прокрутки, используя для этого функции *CWnd::OnHScroll* и *CWnd::OnVScroll*. Во избежание такой, достаточно типичной, ситуации в библиотеке классов MFC реализован специальный класс — *CScrollView*, который берет на себя всю работу по обеспечению поддержки

автоматической прокрутки и масштабирования. При таком подходе на вашу долю остается совсем немного: нужно рассчитать величину прокрутки, базируясь на размере всего документа, и из переопределенной функции *CView::OnInitialUpdate* или *CView::OnUpdate* вызвать функцию *SetScrollSizes*. Ее вызов устанавливает режим отображения для представления, его полный размер и величины горизонтальной и вертикальной прокрутки. Все размеры даются в логических единицах, которые обычно рассчитываются, исходя из данных, хранящихся в документе.

Вместо прокрутки класс *CScrollView* может автоматически установить шкалу представления относительно текущих размеров окна. В этом режиме представление не имеет полос прокрутки и логическое представление сжато или растянуто точно по размеру рабочей области окна. Для использования этой возможности следует вызвать *CScrollView::SetScaleToFitSize*.[1]

Полоса прокрутки в представлении появляется в трех случаях:

- если для представления установлен стандартный стиль Windows, включающий стили WS_HSCROLL и WS_VSCROLL
- когда во фрейм, содержащий представление добавлен элемент управления "полоса прокрутки"; при этом библиотека посылает сообщения WM_HSCROLL и WM_VSCROLL из фрейма в текущее активное представление
- когда объект *CScrollView* расположен в *CSplitterWnd* с раздельными полосами прокрутки; библиотека посылает сообщения WM_HSCROLL и WM_VSCROLL из *CSplitterWnd* в текущую активную разделенную область.

Рассмотрим основные члены этого класса.

CScrollView::**CScrollView**() —

создает объект класса. До использования этого объекта необходимо вызвать одну из функций — SetScrollSizes или SetScaleToFitSize.

Для работы в режиме прокрутки необходимо, прежде всего, установить размеры представления:

```
void CScrollView::SetScrollSizes(
  int nMapMode,
  SIZE sizeTotal,
  const SIZE &sizePage = sizeDefault,
  const SIZE &sizeLine = sizeDefault) —
```

эта функция служит для регулировки характеристик прокрутки текущего представления. Параметр nMapMode задает текущий режим отображения, и может принимать одно из следующих значений: MM_TEXT, MM_HIMETRIC, MM_TWIPS, MM_HIENGLISH, MM_LOMETRIC и MM_LOENGLISH. Общие размеры представления определены в sizeTotal. Они могут быть как фиксированными, так и зависеть от размера ассоциированного документа. Величина вертикальной и гори-

[1] Нельзя пользоваться обеими функциями *SetScaleToFitSize* и *SetScrollSizes* одновременно.

зонтальной прокрутки "на страницу" задается параметром sizePage, а "на строку" — параметром sizeLine. Обычно, функция вызывается из функции OnUpdate или OnInitialUpdate.

```
void CNoteView::OnUpdate(CView*, LPARAM, CObject *pHint)
{
  if(pHint != NULL)
  {
    // нам ли предназначено сообщение?
    if(pHint->IsKindOf(RUNTIME_CLASS(CNoteView)))
    {
      // параметры размера бумаги хранятся у приложения и
      // надо получить к ним доступ
      CNDApp *pApp = (CNDApp*)AfxGetApp();
      CSize sizeTotal;

      // формируем размеры области вывода, учитывая
      // как размер бумаги, так и поля
      sizeTotal.cx = (pApp->m_sizePaper.cx   -
                      pApp->m_rtMargin.right -
                      pApp->m_rtMargin.left - 120);
      sizeTotal.cy = (pApp->m_sizePaper.cy -
                      pApp->m_rtMargin.bottom -
                      pApp->m_rtMargin.top - 120);

      // в соответствии с текущими размерами бумаги и полей
      // устанавливаем новые параметры прокрутки
      SetScrollSizes(MM_LOMETRIC, sizeTotal);
    }
  }

  // Обновляем все, хотя можно и ограничить размеры области
  Invalidate(TRUE);
}
```

Установленные величины области прокрутки можно в любой момент запросить, используя функции:

CSize CScrollView::**GetTotalSize**() —
позволяет узнать полные текущие горизонтальный и вертикальный размеры (в логических единицах) прокручиваемого представления.

и

void CScrollView::**GetDeviceScrollSizes**(
int &nMapMode,
SIZE &sizeTotal,
SIZE &sizePage,
SIZE &sizeLine) —
возвращает текущие значения режима отображения (параметр nMapMode), полного размера (параметр sizeTotal), а также величины прокрутки на страницу

(параметр sizePage) и на строку (параметр sizeLine). Все размеры даются в единицах устройства, и для их перевода в логические, вне функций OnDraw и OnPrint, следует создать контекст устройства

```
CClientDC dc(this);
```

и преобразовать единицы

```
dc.DPtoLP(sizeTotal);
```

Кроме представленных, в классе реализованы знакомые по работе с элементом управления SCROLL BAR функции получения и установки позиции бегунка полосы прокрутки.

CPoint CScrollView::**GetDeviceScrollPosition**(),

CPoint CScrollView::**GetScrollPosition**() —

вызываются, когда необходимо узнать горизонтальную и вертикальную позиции бегунка в зоне прокрутки. Возвращаемая пара координат соответствует размещению в документе, в котором был перемещен левый верхний угол представления (рис. 31). Они полезны для преобразования позиции мыши в позицию прокручиваемого представления. Первая функция возвращает значения в единицах устройства, а вторая — в логических единицах.

void CScrollView::**ScrollToPosition**(POINT pt) —

устанавливает прокрутку в заданную позицию представления. При этом поле pt.sx должно быть больше, или равно нулю, но меньше полного размера представления. Это же справедливо для поля pt.sy, если режим отображения установлен в MM_TEXT. В противном случае эта величина должна быть отрицательной. Если работа с представлением осуществляется в режиме масштабирования, то функцию вызывать нельзя.

Для перехода в режим масштабирования достаточно вызвать функцию

void CScrollView::**SetScaleToFitSize**(SIZE sizeTotal)

которая осуществляет автоматическое масштабирование области вывода представления относительно размера текущего окна. При использовании прокрутки только часть логического представления может быть видима в текущий момент времени. При использовании масштабирования представление не имеет полос прокрутки, и его логический размер изменяется так, чтобы точно совпасть со всей рабочей областью окна. Если окно изменяет размер, то промасштабированное представление отображает свои данные в новой области (рис. 42). Наиболее часто она используется для реализации операции "Zoom to fit".

Обе эти функции (*SetScrollSizes* и *SetScaleToFitSize*) рекомендуется вызывать в обработчике *OnInitialUpdate*. Очевидно, что перед их использованием должны быть установлены размеры самого окна представления, например, следующим образом:

```
GetParentFrame()->RecalcLayout();
```

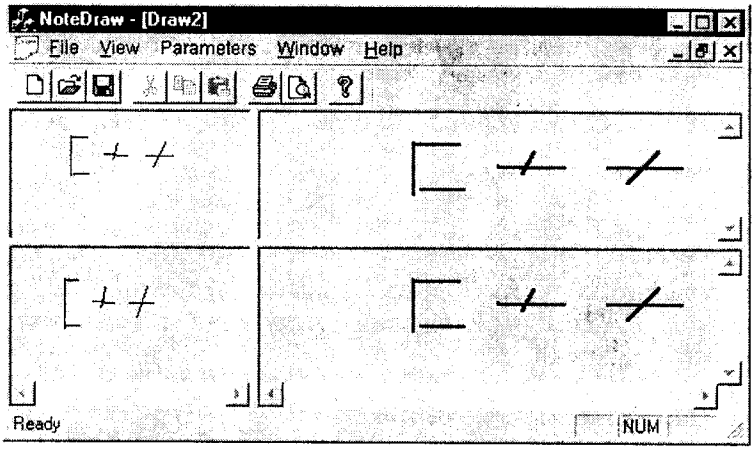


Рис. 42. Разделенное окно представления в режиме масштабирования

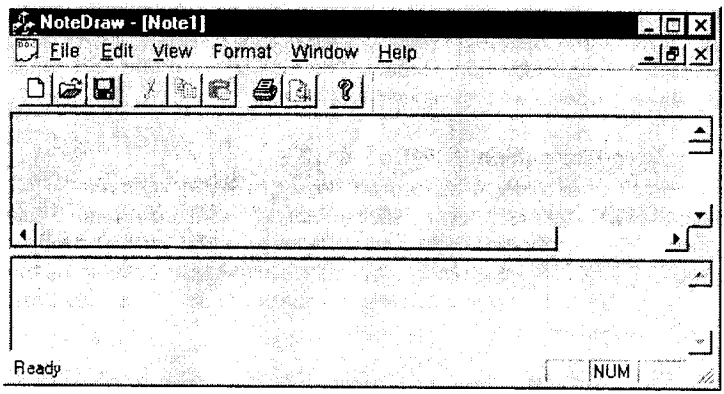


Рис. 43. Фрейм документа до выполнения функции ResizeParentToFit.

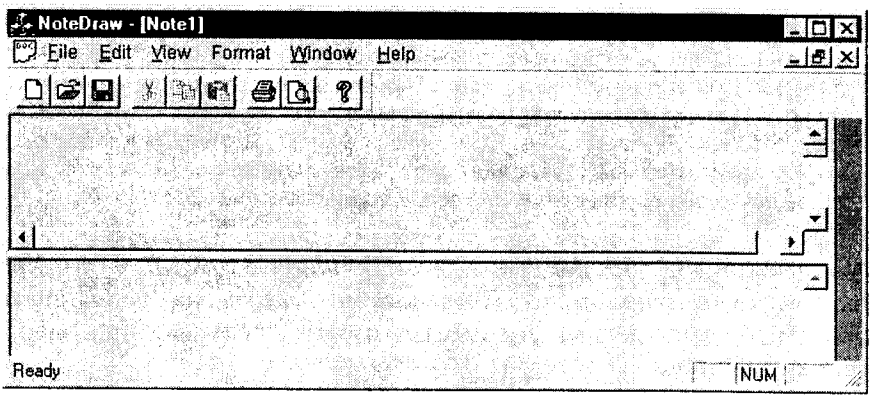


Рис. 44. Фрейм документа после выполнения функции ResizeParentToFit(FALSE).

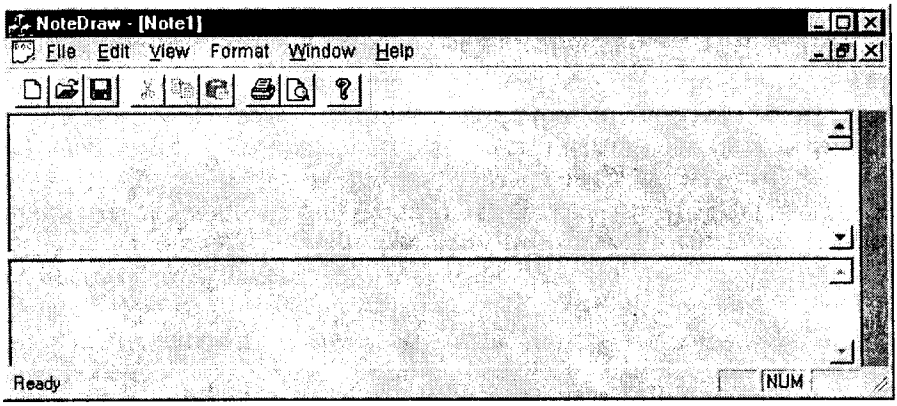


Рис. 45. Фрейм документа после выполнения функции ResizeParentToFit(TRUE).

Для согласования размеров фрейма документа и представления применяется функция

void CScrollView::**ResizeParentToFit**(BOOL bShrinkOnly = TRUE)

которая приводит размеры родительского фрейма в соответствие с размерами представления. Характер изменения размера определяется параметром bShrinkOnly. Если он равен TRUE, то родительский фрейм сжимается до размеров представления; в противном случае представление подгоняется точно под размер фрейма, что может иметь неожиданные последствия в случае с фреймом MDI-приложения или с экраном.

Класс CSplitterWnd

Прокрутка позволяет работать с документом, размер которого больше, чем можно отобразить в окне, но при этом большая часть документа скрыта от пользователя. Вторым вариантом, который мы также рассматривали, является масштабирование. Но если размеры документа достаточно большие, то этот способ хорош только для получения общей информации о его структуре, т. к. при этом на экране трудно рассмотреть его содержимое. Для того чтобы пользователь мог одновременно просматривать различные части документа, в Windows используется так называемое "разделенное окно", которое представляет собой одно окно, разбитое на несколько областей. Такое окно содержит "блок разделителя" (splitter box) в верхней части вертикальной и левой части горизонтальной полос прокрутки, двойное нажатие на который приводит к разделению области, соответственно, на вертикальные или горизонтальные внутренние области. Сами области отделены друг от друга специальной полосой разделения (splitter bar), которая служит также для перераспределения площади окна между соседними областями. Место пересечения горизонтальной и вертикальной полос прокрутки (splitter intersection) позволяет одновременно регулировать и вертикальный и горизонтальный размеры областей.

Для поддержки разделенных окон в библиотеке классов MFC реализован специальный класс *CSplitterWnd*. Каждая область представляет собой отдельное окно, которое управляется объектом *CSplitterWnd* и обычно является определенным объектом приложения, производным от *CView* (или его производного класса), но может быть и произвольным объектом *CWnd*, который имеет соответствующий идентификатор дочернего окна. При этом класс, на основе которого создается область, должен использовать макросы DECLARE_DYNCREATE и IMPLEVENT_DYNCREATE для поддержки режима динамического создания. Очевидно, что для областей гораздо легче (и лучше) использовать классы представлений, чем "обычные" оконные классы. В зависимости от того, каким образом — по горизонтали или по вертикали, разбито окно, для соответствующих областей используются названия "строка" и "столбец". Исходное окно при этом идентифицируется как "нулевая строка" и "нулевой столбец".

Класс *CSplitterWnd* поддерживает два различных стиля разделенных окон: статические и динамические.

В статических разделенных окнах (рис. 46) области создаются только один раз, в момент создания самого окна. Их число и порядок расположения не могут быть изменены в процессе работы. Полосы разделителей используются для изменения размеров различных областей, которые обычно имеют различные классы представлений.

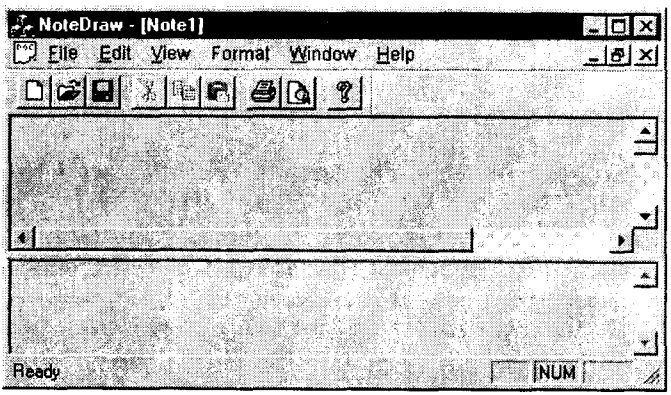


Рис. 46. Статическое разделенное окно с двумя горизонтальными областями

В динамических разделенных окнах (рис. 47) дополнительные области могут создаваться и разрушаться в произвольные моменты времени. Эти окна создаются с единственным представлением, и полосы разделителей предоставляют начальное разделение. Если представление разделено в одном направлении, то дополнительный объект представления динамически создается в представленной новой области. Если представление разделяется в двух направлениях, то три новых представления создаются в трех новых областях. Когда разделитель активен, блок разделителя изображается в виде

разделительной полосы между областями. Дополнительные объекты представления разрушаются, когда пользователь удаляет разделитель, но начальное представление (нулевая строка, нулевой столбец) остается, пока не будет разрушено его окно.

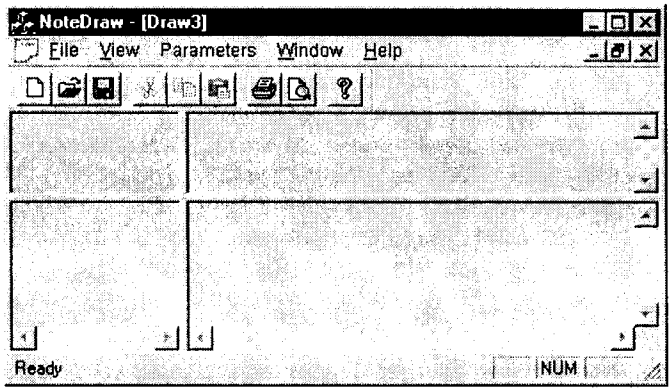


Рис. 47. Динамическое разделенное окно с двумя горизонтальными и двумя вертикальными областями

При создании любого из этих разделенных окон, необходимо, прежде всего, определить максимальное число строк и столбцов, которыми разделитель будет управлять. Для статического разделенного окна области должны быть созданы и все заполнены строками и столбцами. Для динамического разделяемого окна первая область создается автоматически при создании объекта *CSplitterWnd*.

В обоих случаях число областей, на которые можно разделять окно, ограничено: для статических разделяемых окон — 16 строк и 16 столбцов; для динамических — 2 строки и 2 столбца. Одна из многочисленных рекомендаций фирмы Microsoft касается и конфигураций разделенных окон:

строка/столбец	**рекомендация фирмы Microsoft по использованию**
	для статических окон
1 строка × 1 столбец	обычно областями, основанными на различных объектах
2 строки × 1 столбец	
2 строки × 2 столбца	обычно областями, основанными на одинаковых объектах
	для динамических окон
1 строка × 2 столбца	для данных, расположенных в колонках
2 строки × 1 столбец	для текстовых или других данных
2 строки × 2 столбца	для сеточных номограмм и табличных данных

Для каждой строки, как и для каждого столбца, существуют минимальные размеры, и если какая-либо область имеет размеры меньшие, чем минимальные, то соответствующая область не отображается на экране. Для статических разделяемых окон эти значения равны 0, а для динамических определяются соответствующим параметром, задаваемым при создании окна этой области. Эти минимальные размеры в дальнейшем могут быть изменены.

Как и многие другие окна Windows, окно класса *CSplitterWnd* может иметь полосы прокрутки, которые определяются либо указанием соответствующего стиля окна при его создании — WS_HSCROLL или WS_VSCROLL, либо могут представлять собой дочерний элемент управления. В этом плане они ничем не отличаются от других типов окон. Но наряду с этим, класс *CSplitterWnd* поддерживает специальный тип полос прокрутки, которые могут использоваться совместно различными областями. В этом случае, когда пользователь перемещает бегунок полосы прокрутки, соответствующее сообщение посылается во все области.

Полосы прокрутки создаются при создании разделяемого окна. Например, объект класса *CSplitterWnd*, который имеет одну строку, два столбца и стиль WS_VSCROLL, изображает вертикальную полосу прокрутки, которая используется совместно двумя областями, и сообщение WM_VSCROLL посылается в обе.

Теперь, после небольшого экскурса в область разделяемых окон, перейдем к рассмотрению тех возможностей, которые можно использовать для реализации перечисленных свойств.

Обычно объект класса *CSplitterWnd* внедряется в свой родительский объект класса *CFrameWnd* для SDI- и *CMDIChildWnd* для MDI-приложений. В этом случае вызывается конструктор класса:

CSplitterWnd::**CSplitterWnd**() —
создает объект соответствующего класса.

Поскольку, как мы выяснили, окно этого объекта тесно связано с рабочей областью родительского окна, то, прежде всего, необходимо переопределить функцию родительского фрейма *CFrameWnd::OnCreateClient*, из которой вызвать одну из двух функций, отвечающих за создание окна Windows и присоединение его к соответствующему объекту.

```
BOOL CSplitterWnd::Create(
  CWnd *pParentWnd,
  int nMaxRows,
  int nMaxCols,
  SIZE sizeMin,
  CCreateContext *pContext,
  DWORD dwStyle = WS_CHILD | WS_VISIBLE | WS_VSCROLL | WS_HSCROLL |
                  SPLS_DYNAMIC_SPLIT,
  UINT nID = AFX_IDW_PANE_FIRST) —
```

создает динамическое разделяемое окно, которое обычно используется для создания и прокрутки определенного числа индивидуальных областей (или

представлений) того же самого документа. Все области, используемые в таком окне, должны быть одного класса. При успешном создании окна Windows функция возвращает TRUE, в противном случае — FALSE. В качестве параметров она использует: pParentWnd — указатель на родительский фрейм; nMaxRows и nMaxCols — максимальное число строк и столбцов, которое может иметь разделенное окно (оба эти параметра не могут быть больше 2); sizeMin — минимальная высота строки или ширина столбца, при котором область будет изображаться на экране; pContext — указатель на структуру CCreateContext, который в большинстве случаев является соответствующим параметром родительского фрейма или может быть равен NULL; nID — идентификатор дочернего окна, в качестве которого может использоваться значение, указанное по умолчанию, если это окно не используется внутри другого разделяемого окна; dwStyle — стиль окна, в дополнение к стандартным оконным стилям, используется SPLS_DYNAMIC_SPLIT, наличие которого указывает на то, что создается динамическое разделяемое окно.

```
BOOL CDrawFrame::OnCreateClient(LPCREATESTRUCT lpcs,
                                CCreateContext* pContext)
{
  // создаем динамическое разделяемое окно с двумя строками и
  // двумя столбцами, устанавливая некоторые начальные
  // минимальные размеры, чтобы сначала представление занимало
  // все разделяемое окно
  return m_splitWnd.Create(this, 2, 2, CSize(10, 10), pCon-
text);
}
```

BOOL CSplitterWnd::**CreateStatic**(
  CWnd *pParentWnd,
  int nRows,
  int nCols,
  DWORD dwStyle = WS_CHILD | WS_VISIBLE,
  UINT nID = AFX_IDW_PANE_FIRST) —

создает статическое разделяемое окно, в котором пользователь может изменять только размер индивидуальных областей, но не их число или порядок. Количество областей задается параметрами nRows и nCols, значения которых не должны быть больше 16. Функция автоматически устанавливает минимальные высоту строки и ширину столбца в нулевое значение. Для добавления в это окно полос прокрутки необходимо дополнительно установить стили WS_VSCROLL и WS_HSCROLL к значению по умолчанию параметра dwStyle. При успешном создании окна Windows функция возвращает TRUE, в противном случае — FALSE.

В отличие от использования функции *Create*, при создании статического разделяемого окна необходимо самостоятельно позаботиться о создании всех его областей, которые могут принадлежать различным оконным классам. Обычно для этого используется функция *CreateView*, но можно создавать любой оконный класс, не обязательно основанный на представлении. Это следует реализовать до завершения функции *OnCreateClient* родительского фрейма для корректного изображения окна.

BOOL CSplitterWnd::**CreateView**(
int row,
int col,
CRunTimeClass *pViewClass,
SIZE sizeInit,
CCreateContext *pContext) —

создает область статического разделяемого окна. Строка и столбец, в которых будет размещаться вновь создаваемая область, задаются в параметрах row и col. Тип области передается в функцию через pViewClass — указатель на структуру *CRunTimeClass*,[1] определяющую класс представления. Параметр sizeInit устанавливает начальный размер нового представления.

```
BOOL CNoteFrame::OnCreateClient(LPCREATESTRUCT lpcs,
                                CCreateContext* pContext)
{
  // создаем статическое разделяемое окно с двумя строками
  if(!m_spltWnd.CreateStatic(this, 2, 1))
  {
    TRACE0("Ошибка при создании разделенного окна\n");
    return FALSE;
  }

  // создаем представление класса CNoteView в первой строке
  // и отводим под него 2/3 всего разделяемого окна
  if(!m_spltWnd.CreateView(0, 0, pContext->m_pNewViewClass,
      CSize(lpcs->cx, lpcs->cy*2/3), pContext))
  {
    TRACE0("Ошибка при создании представления для вывода\n");
    return FALSE;
  }

  // создаем представление класса CTextView во второй строке
  // и отводим под него оставшуюся часть разделяемого окна
  if(!m_spltWnd.CreateView(1, 0, RUNTIME_CLASS(CTextView),
      CSize(0, 0), pContext))
  {
    TRACE0("Ошибка при создании представления для ввода\n");
    return FALSE;
  }

  // устанавливаем фокус на представление CTextView,
  // отвечающее за ввод текста, — на вторую строку
  SetActiveView((CView*)m_spltWnd.GetPane(1, 0));

  return TRUE;
}
```

[1] Подробно структура *CRunTimeClass* была рассмотрена в первом томе.

Еще раз отметим: для динамического разделяемого окна вызов функции *CreateView* осуществляется автоматически, а при создании всех областей статического разделяемого окна необходимо явно вызывать эту функцию.

Рассмотрев процесс создания, перечислим функции, которые позволяют получать информацию и изменять параметры разделяемого окна.

int CSplitterWnd::**GetRowCount**()

и

int CSplitterWnd::**GetColumnCount**() —

позволяют узнать, соответственно, текущее число строк или столбцов в разделяемом окне. Для статического окна это одновременно и максимальное число.

void CSplitterWnd::**GetRowInfo**(
int row,
int &cyCur,
int cyMin)

и

void CSplitterWnd::**GetColumnInfo**(
int col,
int &cxCur,
int cxMin) —

позволяют получить информацию о строке (столбце), определяемой параметром row (col). Через параметр cyCur (cxCur) возвращается информация о текущей, а через cyMin (cxMin) — о минимальной высоте строки (ширине столбца). Все размеры даются в пикселях.

void CSplitterWnd::**SetRowInfo**(
int row,
int &cyIdeal,
int cyMin)

и

void CSplitterWnd::**SetColumnInfo**(
int col,
int &cxIdeal,
int cxMin) —

позволяют установить новые значения минимальной (параметр cyMin (cxMin)) и идеальной (параметр cyIdeal (cxIdeal)) высоты строки (ширины столбца), заданной в параметре row (col). Когда библиотека MFC изображает разделяемое окно, она располагает области в строках и столбцах в соответствии с их идеальными размерами, начиная от левого верхнего до правого нижнего угла его рабочей области.

CWnd* CSplitterWnd::**GetPane**(
int row,
int col) —

возвращает указатель на оконный объект, ассоциированный с областью, которая задана параметрами row и col. Обычно это класс, производный от CView.

BOOL CSplitterWnd::**IsChildPane**(
CWnd *pWnd,
int &row,
int &col) —

проверяет, является ли объект класса pWnd дочерней областью текущего разделяемого окна. Через параметры row и col возвращаются значения строки и столбца найденной области. Если соответствующая область не найдена, то функция возвращает FALSE, а в row и col записываются нулевые значения.

int CSplitterWnd::**IdFromEowCol**(
int row,
int col) —

позволяет получить идентификатор дочернего окна для области, заданной своими строкой row и столбцом col. Эта функция используется в случае, если при создании соответствующей области применялось "обычное", а не производное от CView, окно.

void CSplitterWnd::**RecalcLayout**() —

обновляет изображение разделяемого окна после того, как были изменены размеры строк(и) и/или столбцов(а). Функцию следует применять в случае, если окно уже видимо. Библиотека MFC вызывает ее, когда пользователь изменил размер окна или его области.

DWORD CSplitterWnd::**GetScrollStyle**() —

возвращает текущий стиль совместно используемых полос прокрутки для разделяемых окон. Возможны следующие значения: WS_HSCROLL и WS_VSCROLL. Если окно не имеет совместно используемых полос прокрутки, то возвращается нуль.

void CSplitterWnd::**SetScrollStyle**(DWORD dwStyle) —

устанавливает новый стиль полосы прокрутки для разделяемого окна, которое поддерживает совместно используемые полосы прокрутки. Если функция вызывается без установленного стиля, то полоса прокрутки убирается с экрана, но не удаляется. После выполнения этой функции необходимо вызвать функцию RecalcLayout для перерисовки разделяемого окна.

Следующая группа функций может быть переопределена в производных классах для реализации индивидуальных возможностей.

virtual void CSplitterWnd::**OnDrawSplitter**(
CDC *pDC,
ESplitType nType,
const CRect &rect) —

вызывается библиотекой MFC для отображения и установки точных характеристик разделяемого окна. Параметр pDC задает контекст устройства, и если он равен NULL, то вызывается функция CWnd::RedrawWindow, и рисование производится для неразделяемого окна. Область, которая перерисовывается, определяется параметром rect. Параметр nType может принимать одно из следующих значений:

splitBox	блок разделителя
splitBar	полоса, которая изображается между двумя разделенными областями

splitIntersection	точка пересечения разделенных областей; не используется для Windows 95
splitBorder	границы разделенного окна

Если образ элемента по каким-либо причинам вас не устраивает, то для изображения своего варианта необходимо переопределить эту функцию.

virtual void CSplitterWnd::**OnInvertTracker**(const CRect &rect) —
вызывается библиотекой MFC во время изменения размеров областей разделяемого окна. Параметр rect определяет отслеживаемый прямоугольник.

virtual void CSplitterWnd::**DeleteView**(
int row,
int col) —
вызывается библиотекой MFC, когда представление было удалено из динамического разделяемого окна. Если удаленное представление было активно, то активным становится следующее представление. Параметры row и col определяют область (по строке и столбцу), в которой удаляется представление.

virtual BOOL CSplitterWnd::**SplitRow**(int cyBefore)
и
virtual BOOL CSplitterWnd::**SplitColumn**(int cxBefore) —
вызываются библиотекой MFC при создании горизонтальной (вертикальной) области в динамическом разделяемом окне. Параметр cyBefore (cxBefore) определяет позицию (в пикселях), в которой производится разделение.

virtual void CSplitterWnd::**DeleteRow**(int rowDelete)
и
virtual void CSplitterWnd::**DeleteColumn**(int colDelete) —
вызываются библиотекой MFC, когда из динамического разделяемого окна *была* удалена строка rowDelete (столбец colDelete).

virtual CWnd* CSplitterWnd::**GetActivePane**(
int *pRow = NULL,
int *pCol = NULL) —
используется для определения параметров (номера строки — pRow и столбца — pCol) активной области в разделяемом окне. Если таковой нет, функция возвращает NULL.

virtual void CSplitterWnd::**SetActivePane**(
int row,
int col,
CWnd *pWnd = NULL) —
устанавливает фокус на соответствующую область внутри фрейма. Активизируемая область определяется либо номером строки row и столбца col, если указатель на оконный объект pWnd равен NULL, либо непосредственно самим указателем. Эта функция вызывается библиотекой MFC, когда пользователь изменил фокус активной области, но может быть вызвана и непосредственно для принудительной смены фокуса.

virtual BOOL CSplitterWnd::**CanActivateNext**(BOOL bPrev = FALSE) —
проверяет возможность выполнения команд Next Pane (Следующее окно) или Previous Pane (Предыдущее окно). Команда определяется параметром bPrev. Если он равен TRUE, то проверяется предшествующая, если FALSE — следующая область.

virtual void CSplitterWnd::**ActivateNext**(BOOL bPrev = FALSE) —
активизирует следующую (bPrev = FALSE) или предшествующую (bPrev =TRUE) область разделенного окна.

virtual BOOL CSplitterWnd::**DoKeyboardSplit**() —
выполняет команду клавиатуры, обычно Window Split (Разбить окно), на разделение окна. При успешном завершении возвращает TRUE.

virtual BOOL CSplitterWnd::**DoScroll**(
CView *pViewFrom,
UINT nScrollCode,
BOOL bDoScroll = TRUE) —
вызывается библиотекой MFC для выполнения синхронизации прокрутки разделенных окон, когда представление получает сообщение о прокрутке. Параметр pView определяет указатель на объект представления, откуда поступило сообщение. Надо ли действительно осуществлять прокрутку, определяется в параметре bDoScroll. Если он равен TRUE, то прокрутку необходимо выполнять, если FALSE, то прокрутка не производится. Последнее имеет место, если, например, соответствующее представление не имеет полосы прокрутки. Параметр nScrollCode определяет код полосы прокрутки и состоит из двух частей: младший байт задает тип прокрутки по горизонтали, а старший — по вертикали.

virtual BOOL CSplitterWnd::**DoScrollBy**(
CView *pViewFrom,
CSize sizeScroll,
BOOL bDoScroll = TRUE) —
вызывается библиотекой MFC в ответ на сообщение о прокрутке и выполняет синхронизированную прокрутку разделенных окон на величину sizeScroll, заданную в пикселях. Положительное значение определяет направление прокрутки вниз и вправо, а отрицательное — вверх и влево.

virtual BOOL CSplitterWnd::**CreateScrollBarCtrl**(
DWORD dwStyle,
UINT nID) —
создает разделяемый элемент управления "полоса прокрутки". Параметр dwStyle определяет его оконный стиль, а *nID* — идентификатор дочернего окна, куда этот элемент управления будет вставлен; он может иметь значение AFX_IDW_PANE_FIRST, если заданное разделенное окно не вложено в другое разделенное окно.

На этом мы заканчиваем общее рассмотрение классов документов и представлений и переходим к достаточно интересной теме — печать и предварительный просмотр документа перед печатью.

Глава 28

Печать и предварительный просмотр документов

Для взаимодействия приложений с графическими устройствами, такими как экран дисплея, принтер или плоттер, в Windows используется интерфейс графических устройств (GDI). И когда приложению требуется выполнить операцию вывода графического изображения, оно осуществляет это посредством GDI и работает не с реальным (физическим), а с логическим устройством. Выполняя запрос приложения, GDI обращается к драйверу соответствующего устройства вывода, который уже непосредственно работает с физическим устройством. Такой подход позволяет приложениям Windows единообразно работать на любом оборудовании, лишь бы был установлен соответствующий драйвер. Само приложение обращается к устройству вывода посредством контекста устройства, который представляет собой структуру данных, содержащую информацию о том, как нужно выполнять операции вывода на данном устройстве. Эта структура при своем создании ассоциируется с драйвером конкретного физического устройства, преобразующего все направляемые ему вызовы в такие, которые им поддерживаются.[1]

Все сказанное в полной мере относится и к выводу на принтер: прежде всего, необходимо создать контекст отображения, связанный с принтером, куда затем и направлять весь графический вывод (как и при выводе на экран). Однако отличия все-таки есть. И связаны они, прежде всего, с тем, что при печати графический вывод осуществляется постранично, а листы бумаги могут иметь различные размеры и/или ориентацию (портретную или ландшафтную). За отслеживание этих параметров целиком отвечает программист. Есть и некоторые другие отличия. Но от слов пора переходить к делу.

Первое, что следует совершенно четко представлять, что вывод на принтер осуществляется исключительно через буфер, который, однако, не просто служит для временного хранения данных. Все несколько сложнее. Дело в том, что при вызове функций рисования для контекста принтера, эти ко-

[1] Подробно о контекстах устройств было рассказано в главе 3.

манды GDI выполняются не сразу, а накапливаются в специальном метафайле. И только после того, как приложение завершит рисование одной страницы документа, созданный метафайл "проигрывается" в контексте принтера. Печать происходит именно в этот момент. Сам механизм скрыт от приложения, и единственное, что требуется — это сообщить GDI о начале и завершении процесса печати листа.

Вторым важным моментом является то, что печать в системе Windows организована через специальную очередь, которая организуется приложением Print Manager (Диспетчер печати). Для печати документа приложения помещают в эту очередь свои данные (задания на печать), которые будут выводиться на принтер в порядке поступления, в фоновом режиме.

Для печати документов Win32 API предоставляет следующие функции:

- *StartDoc* – формирует задание на печать нового документа;
- *StartPage* – подготавливает контекст устройства вывода для печати новой страницы — готовит метафайл; необходимо вызвать эту функцию перед выводом в контекст устройства;
- *EndPage* – завершает программный процесс печати одной страницы — формирование метафайла, после чего он выводится непосредственно на принтер;
- *EndDoc* – завершает процесс печати документа;
- *AbortDoc* – служит для принудительного завершения процесса печати;
- *SetAbortProc* – используется для обеспечения возможности фоновой печати и принудительного завершения процесса печати;
- *ResetDC* – позволяет настроить индивидуальные параметры печати отдельных листов документа.

Как видите, функций совсем не много. А это значит, что всю основную работу необходимо выполнить самостоятельно. Сам же процесс графического вывода на принтер достаточно прост.[1] Прежде всего, формируется контекст отображения для принтера. Затем необходимо обеспечить возможность принудительного (аварийного) завершения печати, заполнить структуру *DOCINFO* и вызвать функцию *StartDoc*. После завершения подготовительных действий организуется цикл печати страниц документа, где должны быть проверены специально созданные флаги отмены и завершения печати. Вывод на принтер в Windows организуется постранично, поэтому перед началом печати каждой страницы необходимо вызвать функцию *StartPage*, которая устанавливает параметры специального метафайла, куда будет направляться вывод для всей страницы. После того как вся страница записана в метафайл, следует закрыть его и отправить непосредственно на принтер.

---

[1] Пока мы описываем этот процесс в упрощенном виде, не вдаваясь в некоторые детали, например, каким образом определять окончание страницы или как настроить параметры принтера.

Для этой цели используется функция *EndPage*. Цикл выполняется для всех страниц и завершается либо аварийно, либо после вывода всего документа. В первом случае следует вызвать функцию *AbortDoc*, а во втором – *EndDoc*. И в заключение необходимо освободить используемый контекст устройства. Подобный процесс, с теми или иными вариациями, выполняется при любом выводе документа на печать.

После того как сложилось общее представление об организации процесса печати в Windows, вернемся к библиотеке классов MFC и рассмотрим, как в ней реализован вывод данных документа на принтер, и другие связанные с этим вопросы. И начнем с выбора и настройки параметров принтера.

Выбор и настройка параметров принтера

Поскольку вся информация об установленных принтерах хранится в самой операционной системе, то вполне логичным выглядит то, что за рассматриваемый процесс отвечает приложение. В классе *CWinApp* реализован специальный обработчик стандартной команды ID_FILE_PRINT_SETUP:

afx_msg void CWinApp::**OnFilePrintSetup**()

который выводит на экран стандартный блок диалога Print Setup (Настройка печати) для настройки текущей конфигурации принтера.

Для подключения обработчика к команде, в карту сообщений объекта "приложение" необходимо добавить оператор

```
ON_COMMAND(ID_FILE_PRINT_SETUP, CWinApp::OnFilePrintSetup)
```

В большинстве случаев вполне достаточно этого стандартного обработчика, но, при необходимости, можно создать и что-нибудь свое. Изобретать велосипед мы не будем, а для иллюстрации рассмотрим лишь, каким образом можно реализовать индивидуальную настройку стандартного блока диалога Print Setup (Настройка печати), класс для работы с которым был нами описан в главе 15. Прежде всего, в карту сообщений класса приложения добавляем оператор:[1]

```
ON_COMMAND(<ID_PRINT_SETUP>, <OnPrintSetup>)
```

Затем в описании класса регистрируем этот обработчик:

```
class CNDApp : public CWinApp
{
...
  //{{AFX_MSG(CNDApp)
  ...
```

[1] Имена в угловых скобках могут быть произвольными.

```
  afx_msg void OnPrintSetup();
  ...
  //}}AFX_MSG
...
};
```

И, наконец, реализуем его. В данном примере при инициализации блока диалога устанавливаем размер бумаги Letter 8 ½ ×11 in и альбомную ориентацию (рис. 48).

```
void CNDApp::OnPrintSetup()
{
  // создаем объект "стандартный блок диалога"
  // Print Setup (Настройка печати)
  CPrintDialog dlgPD(TRUE);

  // для получения параметров принтера, установленного
  // по умолчанию, создаем объект
  PRINTDLG pd;

  // устанавливаем указатель на глобальный блок памяти в NULL
  // с тем, чтобы запросить у системы параметры принтера
  pd.hDevMode = NULL;

  // получаем текущую конфигурацию принтера
  GetPrinterDeviceDefaults(&pd);

  // поскольку параметр pd.hDevMode указывает на глобальный
  // перемещаемый блок памяти, то перед началом работы с ним
  // необходимо его зафиксировать, иначе ничего не получится
  DEVMODE *dm = (DEVMODE *)LocalLock(pd.hDevMode);

  // теперь можно заполнять поля структуры DEVMODE значениями,
  // необходимыми для инициализации блока диалога
  // Print Setup (Настройка печати)
  dm->dmFields = DM_ORIENTATION | DM_PAPERSIZE;
  dm->dmOrientation = DMORIENT_LANDSCAPE;
  dm->dmPaperSize = DMPAPER_LETTER;

  // передаем указатель на заполненную структуру DEVMODE
  // объекту класса CPrintDialog
  dlgPD.m_pd.hDevMode  = pd.hDevMode;

  // теперь зафиксированный блок памяти можно освободить
  LocalUnlock(pd.hDevMode);

  // выводим блок диалога на экран
  if(dlgPD.DoModal() == IDOK)
  {
    // здесь можно зафиксировать выбор пользователя
    ...
  }
}
```

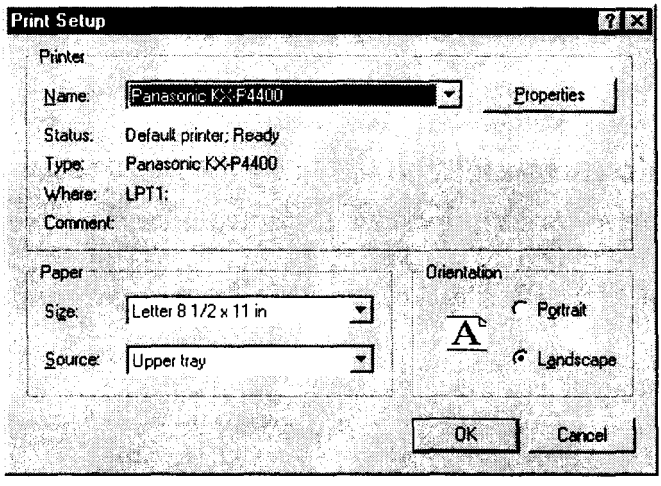


Рис. 48. Стандартный блок диалога Print Setup (Настройка печати)

Возможностей для индивидуальных настроек достаточно много, но, повторяем, библиотека MFC достаточно хорошо выполняет значительную часть работы, и в большинстве случаев нет необходимости вмешиваться в этот процесс.

Для того чтобы узнать текущую конфигурацию принтера, мы воспользовались функцией

BOOL CWinApp::**GetPrinterDeviceDefaults**(PRINTDLG *pd)

которая возвращает информацию либо о принтере, заданном по умолчанию, либо о последней конфигурации принтера, установленной с помощью блока диалога Print Setup (Настройка принтера). Параметр *pd* указывает на структуру *PRINTDLG*, которая служит для настройки параметров блоков диалога Print (Печать) и Print Setup (Настройка печати), и подробно была описана в первом томе (глава 15):

```
...
PRINTDLG pd;
...
// Запрашиваем параметры текущего принтера или
// установленного по умолчанию
GetPrinterDeviceDefaults(&pd);
...
```

Эта часть работы, конечно же, достаточно важна, и обеспечить ее поддержку необходимо, но библиотека MFC делает это прекрасно, а нас в основном интересует дальнейший процесс. Итак, принтер выбран, его параметры установлены – можно печатать. И первоочередной задачей является, естественно, создание контекста устройства для выбранного принтера.

Создание контекста устройства

Библиотека MFC и здесь приходит на помощь – у класса приложения (*CWinApp*) имеется специальная функция:

BOOL CWinApp::**CreatePrinterDC**(CDC &dc)

которая создает контекст устройства (параметр *dc*) для выбранного в текущий момент принтера.

Перед тем как выводить что-либо на печать, достаточно вызвать эту функцию:

```
...
CDC dc;
// Создаем контекст устройства для принтера
AfxGetApp()->CreatePrinterDC(&dc);
// теперь можно выводить информацию в контекст устройства dc
...
```

Вот и все. Не нужно каждый раз выполнять "утомительные" настройки специальных структур, как это приходится делать при использовании Win32 API.

Для того чтобы полностью разобраться в рассматриваемом вопросе, нам осталось только рассмотреть организацию собственно процесса печати.

Печать документов и библиотека MFC

У некоторых читателей мог возникнуть вполне естественный вопрос: "Почему печать документов рассматривается именно в этой части, которая посвящена архитектуре документ/представление?". Дело в том, что в классах представлений реализована достаточно мощная поддержка этого процесса, которая избавит вас от многих дополнительных трудностей, освободив время для решения конкретных задач.

В процессе печати основная роль принадлежит объектам двух классов библиотеки MFC – контексту устройства и представлению, которые тесно взаимодействуют между собой. На рис. 49 представлена схема цикла печати, реализованная в библиотеке классов MFC.

Как видите, представленный цикл очень напоминает тот, который используется в Windows для любого вывода на печать: на экран выводится блок диалога Print (Печать), создается контекст устройства для принтера и вызывается функция *StartDoc* объекта *CDC*. Далее для каждой страницы документа библиотека вызывает функцию *StartPage* контекста принтера, осуществляет вывод данных в метафайл (функция *OnPrint* объекта "представление"), и

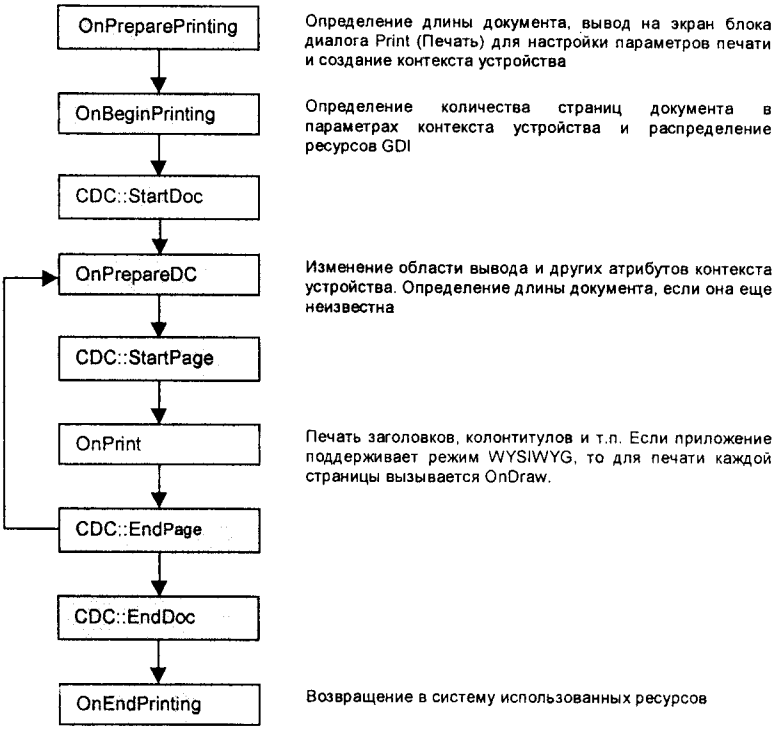


Рис. 49. Цикл печати библиотеки MFC

завершает печать страницы вызовом функции *EndPage* объекта CDC. Когда весь документ выведен на принтер, библиотека вызывает функцию *EndDoc*, завершая цикл печати. Разница заключается только в том, что как сам цикл, так и все участвующие здесь функции, уже реализованы. Наряду с этим, благодаря механизму виртуальных функций, программисту предоставляются широкие возможности для "более тонкой" настройки всего процесса печати. Все эти функции (*OnPreparePrinting*, *OnBeginPrinting*, *OnPrepareDC*, *OnPrint* и *OnEndPrinting*) являются виртуальными, определены в классе *CView* и при их переопределении необходимо соблюдать логику, которая заложена в библиотеку классов MFC. Но сначала рассмотрим сами эти функции:

virtual BOOL CView::**OnPreparePrinting**(CPrintInfo *pInfo) —
вызывается библиотекой классов MFC до начала процесса печати или предварительного просмотра документа. Это самый удобный момент для определения длины документа и настройки параметров печати. Для реализации последней возможности необходимо просто вызвать функцию DoPreparePrinting:

```
BOOL CView::OnPreparePrinting(CPrintInfo *pInfo)
{
  return DoPreparePrinting(pInfo);
}
```

В качестве параметра функция принимает указатель на структуру *CPrintInfo*, которая хранит информацию о задачах печати или предварительного просмотра:[1]

```
struct CPrintInfo
{
  CPrintInfo();               // конструктор
  ~CPrintInfo();              // деструктор

  // указатель на объект "блок диалога" Print (Печать)
  CPrintDialog *m_pPD;

  BOOL m_bPreview;            // TRUE, если предварительный просмотр
  BOOL m_bDirect;             // TRUE, если без отображения на экране
                              // блока диалога Print (Печать)

  BOOL m_bContinuePrinting;   // FALSE для прерывания печати
  UINT m_nCurPage;            // номер текущей печатаемой страницы
  UINT m_bNumPreviewPages;    // число просматриваемых страниц
  CString m_strPageDesc;      // номер изображаемой страницы
  LPVOID m_lpUserData;        // указатель на данные пользователя
  CRect m_rectDraw;           // прямоугольник, определяющий текущую
                              // используемую область страницы

  void SetMinPage(UINT nMinPage); // устанавливает номер первой
                                  // печатаемой страницы документа
  void SetMaxPage(UINT nMaxPage); // устанавливает номер последней
                                  // печатаемой страницы документа
  UINT GetMinPage() const;        // получает номер первой
                                  // печатаемой страницы документа
  UINT GetMaxPage() const;        // получает номер последней
                                  // печатаемой страницы документа
  UINT GetFromPage() const;       // получает номер первой
                                  // печатаемой страницы диапазона
  UINT GetToPage() const;         // получает номер последней
                                  // печатаемой страницы диапазона
};
```

Объект этой структуры используется для обмена информацией между библиотекой и классом представления во время процесса печати. Он создается при выборе команд ID_FILE_PRINT_PREVIEW, ID_FILE_PRINT_DIRECT или ID_FILE_PRINT и разрушается после их выполнения.

Если возникнет необходимость инициализации блока диалога Print (Печать) значениями, отличными от заданных по умолчанию, то лучше всего это сделать именно здесь (до вызова функции *DoPreparePrinting*).

---

[1] Режим предварительного просмотра документа перед печатью будет рассматриваться в этой же главе.

BOOL CView::**DoPreparePrinting**(CPrintInfo *pInfo) —
возвращает TRUE, если можно начинать печать или предварительный просмотр документа, и FALSE в противном случае. Поведение этой функции зависит от того, вызывается ли она для печати документа или для его предварительного просмотра, что определяется параметром m_bPreview структуры, на которую указывает pInfo. В первом случае на экран выводится блок диалога Print (Печать). После его закрытия функция создает контекст устройства на основе установок, сделанных пользователем в этом блоке диалога для принтера, который будет использоваться при печати документа. Если функция вызвана для предварительного просмотра документа, то контекст устройства создается на основе текущих установок принтера.

Для осуществления всех корректных установок, в переопределенной версии этой функции обязательно следует вызвать функцию базового класса *CView::DoPreparePrinting*.

В приведенном ниже фрагменте кода демонстрируется принцип инициализации блока диалога Print (Печать) в зависимости от того, имеется ли во введенном тексте выделенный фрагмент (рис. 50).

```
BOOL CNoteView::OnPreparePrinting(CPrintInfo* pInfo)
{
  // документ не будет иметь больше одной страницы
  pInfo->SetMaxPage(1);

  // перемещаемся по списку представлений
  CNoteDoc* pDoc = (CNoteDoc*)GetDocument();
  POSITION pos = pDoc->GetFirstViewPosition();
  pDoc->GetNextView(pos);

  // получаем указатель на представление,
  // отвечающее за ввод текста (CTextView)
  CTextView *pText = (CTextView *)pDoc->GetNextView(pos);

  CString str;

  // проверяем наличие выделенного фрагмента
  pText->GetSelectedText(str);
  if(!str.IsEmpty())
  {
    // если есть выделенный фрагмент, то переключатель
    // Print range (Печатать) устанавливаем в положение
    // Selection (Выделенный фрагмент)
    CPrintDialog *pPD = pInfo->m_pPD;
    pPD->m_pd.Flags &= ~PD_NOSELECTION;
    pPD->m_pd.Flags |= PD_SELECTION;
  }

  // если выделенного фрагмента нет, то используем
  // установки по умолчанию
...
```

```
  // не забываем вызвать функцию базового класса
  // для проведения корректной инициализации
  return DoPreparePrinting(pInfo);
}
```

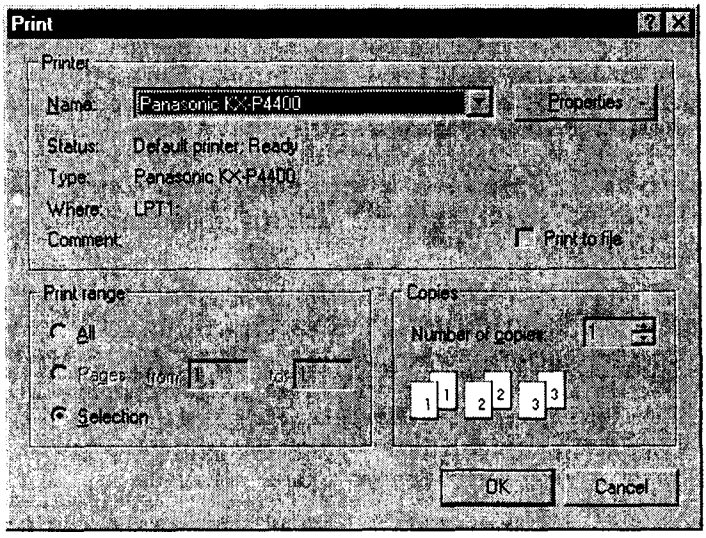


Рис. 50. Стандартный блок диалога Print (Печать)

Если необходим предварительный доступ к контексту устройства, то это можно сделать, переопределив функцию

```
virtual void CView::OnBeginPrinting(
   CDC *pDC,
   CPrintInfo *pInfo)
```

которая вызывается библиотекой классов MFC перед началом процесса печати или предварительного просмотра после того, как была вызвана функция *OnPreparePrinting*. Эта функция используется для размещения необходимых для печати ресурсов GDI, таких как карандаши или шрифты. Сами объекты GDI выбираются в контекст устройства в функции *OnPrint* отдельно для каждой страницы, которая их использует. Если одно и то же представление используется для вывода на экран и на принтер, рекомендуется использовать различные переменные для ресурсов GDI, необходимых для каждого изображения, что позволит обновлять их на экране непосредственно во время процесса печати. Кроме того, эту функцию можно использовать для выполнения инициализации, зависящей от свойств контекста устройства принтера, например, от числа печатаемых страниц. В качестве параметров используются указатели на контекст устройства (параметр *pDC*) и на структуру *pPrintInfo*, которая описывает текущее задание печати (параметр *pInfo*).

Для корректного завершения процесса печати или предварительного просмотра документа необходимо использовать функцию

```
virtual void CView::OnEndPrinting(
  CDC *pDC,
  CPrintInfo *pInfo)
```

которая вызывается библиотекой классов MFC после окончания процесса печати или предварительного просмотра. Если в функции *OnBeginPrinting* были задействованы какие-либо ресурсы GDI, то здесь их необходимо вернуть системе.

Если две последние рассмотренные функции используются для документа в целом, то для настройки параметров отдельной страницы разработчики реализовали функцию

```
virtual void CView::OnPrepareDC(
  CDC *pDC,
  CPrintInfo *pInfo = NULL)
```

которая вызывается библиотекой классов MFC до функций *OnDraw* (предназначенной для изображения на экране) или *OnPrint* (используемой для каждой печатаемой или просматриваемой страницы). Если функция вызвана для печати документа, то она проверяет информацию о странице, имеющуюся в структуре, на которую указывает параметр *pInfo*. Если длина документа не определена, она принудительно прерывает печать после того, как будет выведена одна страница. Остановить процесс печати можно принудительно — записав в поле *m_bContinuePrinting* структуры *pInfo* значение FALSE. В качестве параметров используются: *pDC* — указатель на контекст устройства, используемый для представления образа документа и *pInfo* — указатель на структуру *CPrintInfo*, описывающую текущее задание. В режиме печати или предварительного просмотра поле *m_nCurPage* структуры *CPrintInfo* определяет номер печатаемой страницы (если функция вызвана для вывода на экран, этот параметр должен быть установлен в NULL). При переопределении этой функции, прежде всего, следует вызвать обработчик *OnPrepareDC* базового класса. Переопределение этой функции необходимо в следующих случаях:

- настройка атрибутов контекста устройства, необходимых для определенной страницы;
- динамическое определение числа страниц документа непосредственно во время печати (например, при распечатке неизвестного заранее числа записей базы данных);
- посылка на принтер ESC-кодов с использованием функции *Escape* объекта класса *CDC*, на который указывает параметр *pDC*.

Теперь, когда общая структура процесса печати достаточно ясна, самое время рассмотреть, где же осуществляется собственно вывод (или, если угодно, рисование) данных документа. В предыдущей главе подробно рас-

сматривалась функция *OnDraw*, которая в рамках архитектуры "документ/представление" отвечает за вывод на любое устройство, будь то экран, принтер или что-либо другое. Его конкретный тип задается объектом контекста устройства, передаваемого функции в качестве параметра. Например, для вывода на экран она получает в качестве параметра указатель на объект класса *CPaintDC*, а для печати документа — на объект класса *CDC*, ассоциированного с текущим принтером.

Все это, конечно, справедливо, и можно было бы просто переопределить функцию *OnDraw* для организации вывода на принтер, но ведь печать осуществляется на листы бумаги, имеющие вполне определенные размеры, которые, по меньшей мере, необходимо учитывать. И для того, чтобы не перегружать функцию *OnDraw*, разработчики библиотеки для работы с принтером реализовали в базовом классе всех представлений *CView* отдельную функцию:

```
virtual void CView::OnPrint(
  CDC *pDC,
  CPrintInfo *pInfo = NULL)
```

которая вызывается библиотекой классов MFC после вызова функции *OnPrepareDC* для печати или предварительного просмотра каждой страницы документа. Реализация по умолчанию просто вызывает функцию *OnDraw* с контекстом устройства принтера.

Переопределение этой функции необходимо, например, в следующих случаях:

- при печати многостраничных документов;
- для установки различных образов при выводе на принтер и экран, что допустимо, если приложение не поддерживает режима WYSIWYG;
- для реализации изображения колонтитулов и примечаний.

Здесь есть два тонких момента, требующих пояснения. Пока речь идет о печати одной страницы, то все в порядке – можно использовать реализацию по умолчанию, в которой просто вызывается функция *OnDraw*. Но, как вы помните, для вывода на принтер нескольких страниц необходимо организовать цикл печати. И такой цикл действительно организован в обработчике

```
void CView::OnFilePrint()
```

Именно из него осуществляются вызовы всех рассмотренных функций, и, в большинстве случаев, вполне достаточно переопределить те из них, которые необходимы для индивидуальной организации печати документа.

Второй момент, на который следует обратить внимание, связан с принудительным завершением вывода данных на принтер, т. е. с функциями *AbortDoc* и *SetAbortProc* из Win32 API. Как уже отмечалось, нужно установить при помощи функции *SetAbortProc* специальную функцию, которая и будет отслеживать требование на завершение печати, периодически вызывая функ-

цию *CDC::QueryAbort*, сообщающую, можно ли продолжать процесс печати или его нужно срочно завершить.

Всех этих сложностей можно избежать, если оставить функцию *CView::OnFilePrint* в покое. Пусть она делает свое дело, тем более, что у нее это получается хорошо.

Остановившись на последнем варианте (не переопределяя функцию *OnFilePrint*), продолжим рассмотрение процесса печати и того, как он реализован в библиотеке классов MFC.

Начиная по командам ID_FILE_PRINT или ID_FILE_PRINT_DIRECT процесс печати, библиотека MFC вызывает функцию *OnPreparePrinting* вашего объекта "представление", передавая ей в качестве параметра указатель на уже подготовленную структуру *CPrintInfo*. Поскольку в этот момент приложению неизвестно количество страниц печатаемого документа, то значения первой и последней печатаемой страницы устанавливаются, соответственно, в 1 и −1 (0×FFFF). Здесь удобнее всего установить длину документа (число его страниц), вызвав функцию *SetMaxPage* структуры *CPrintInfo*. Второй задачей функции *OnPreparePrinting* является вызов *DoPreparePrinting*, которая выводит на экран блок диалога[1] Print (Печать) и создает контекст устройства для принтера. Если по каким-либо причинам вас не устраивает внешний вид блока диалога или начальные установки принтера, то их можно изменить перед вызовом этой функции. Те параметры, которые пользователь установил в блоке диалога, доступны для использования через поля структуры *CPrintInfo*. Так, например, если пользователь установил номера страниц, которые требуется напечатать, то их можно получить при помощи функций *GetFromPage* и *GetToPage*. Лаконичность последней фразы не должна создавать иллюзию, что все так уж просто. Для того чтобы воспользоваться функциями *GetFromPage* и *GetToPage*, сначала необходимо проверить состояние группы переключателей Page range (Диапазон печати). Функции вернут достоверные значения только в том случае, если установлен переключатель Pages (Страницы).

Следующей функцией, вызываемой библиотекой MFC, является функция *OnBeginPrinting*. Ее наличие в цикле печати свидетельствует о похвальном желании разработчиков библиотеки предоставить программистам возможность именно здесь определиться и получить от системы все необходимые ресурсы GDI. Правда, на практике мало кто использует эту возможность — обычно ресурсы запрашиваются у системы непосредственно в функциях *OnDraw* или *OnPrint*, и там же, кстати говоря, и возвращаются ей обратно.

После того как функция *CDC::StartDoc* сформирует задание на печать, начинается собственно цикл, включающий в себя две интересующих нас функции — *OnPrepareDC* и *OnPrint*, которые библиотека вызывает для каждой печатаемой страницы, сообщая представлению ее номер в поле *m_nCurPage*

---

[1] При выполнении команды ID_FILE_PRINT_DIRECT блок диалога не выводится, а непосредственно начинается процесс печати документа.

структуры *CPrintInfo*. Первая функция служит для решения трех основных задач: определения области вывода (view port), осуществления проверки на конец документа и, при необходимости, посылки на принтер специальных ESC-кодов. Следует также отметить, что эта функция вызывается как для печати документа, так и для отображения его на экране. Различие заключается в первом передаваемом ей параметре – указателе на контекст принтера или экрана, и в том, что при выводе на экран в качестве второго параметра передается NULL. Для самой печати страницы предназначена функция *OnPrint*, поскольку передаваемые ей параметры несут всю необходимую для этого информацию. Однако, особенно если посмотреть на ее реализацию по умолчанию в классе CView:

```
void CView::OnPrint(CDC* pDC, CPrintInfo*)
{
  ASSERT_VALID(pDC);

  OnDraw(pDC);
},
```

приходишь к выводу, что более правильным подходом является такой, при котором эта функция "занимается своим делом", т. е. отвечает за изображение данных, специфичных именно для вывода на принтер, например, для печати колонтитулов и примечаний, оставляя всю остальную часть работы функции *OnDraw*.[1]

Завершается цикл печати вызовом функций *CDC::EndDoc* и *OnEndPrinting*. И если первая функция завершает процесс печати документа, то вторая призвана освободить все использованные при печати ресурсы GDI.[2]

Таким образом, для того чтобы "организовать" процесс печати, используя возможности, предоставляемые библиотекой классов MFC, достаточно переопределить некоторые из функций (или все) *OnPreparePrinting*, *OnBeginPrinting*, *OnPrepareDC*, *OnPrint* и *OnEndPrinting* (правда, и этого не мало — чего стоит только определение длины документа!).

В заключение обсуждения организации вывода документа на принтер рассмотрим пример переопределения перечисленных функций. Подробные комментарии делают излишним более детальное их обсуждение. Скажем лишь, что рассматриваемые фрагменты демонстрируют вывод колонтитула и номера страницы при печати документа.

```
void CNoteView::OnPrint(CDC* pDC, CPrintInfo* pInfo)
{
  CNDApp *pApp = (CNDApp*)AfxGetApp();
```

---

[1] Действительно, зачем дублировать код, если в Windows организация вывода не зависит от конкретного устройства (аппаратно-независимый вывод).

[2] Если вы все-таки последовали рекомендациям и запросили их у системы в функции *OnBeginPrinting*.

```
  // создаем контекст устройства для принтера
  CDC dc;
  pApp->CreatePrinterDC(dc);

  // получаем размеры бумаги
  CSize sizePaper = CSize(dc.GetDeviceCaps(HORZSIZE)*10,
                          dc.GetDeviceCaps(VERTSIZE)*10);

  CRect rect(0, 0, sizePaper.cx, sizePaper.cy);
...
  // учитываем величины полей
  rect.left   += pApp->m_rtMargin.left;
  rect.right  -= pApp->m_rtMargin.right;
  rect.top    -= pApp->m_rtMargin.top;
  rect.bottom += pApp->m_rtMargin.bottom;

  // преобразуем логические координаты в физические
  pDC->LPtoDP(&rect);

  // изменяем параметры прямоугольника рисования
  pInfo->m_rectDraw.left   = rect.left;
  pInfo->m_rectDraw.right  = rect.right;
  pInfo->m_rectDraw.top    = rect.top;
  pInfo->m_rectDraw.bottom = rect.bottom;

  // устанавливаем новые координаты области вывода
  pDC->SetViewportOrg(rect.left, rect.top);

  // получаем название документа
  CString strHeader = GetDocument()->GetTitle();

  // печатаем заголовок
  PrintHeader(pDC, pInfo, strHeader);

  // рисуем данные документа
  OnDraw(pDC);
}

void CNoteView::PrintHeader(CDC *pDC,
                            CPrintInfo *pInfo,
                            CString strHeader)
{
  // устанавливаем выравнивание влево
  pDC->SetTextAlign(TA_LEFT);

  // для учета величины смещения получаем
  // характеристики шрифта
  TEXTMETRIC tm;
  pDC->GetTextMetrics(&tm);

  int y = tm.tmHeight + 15;
```

```
    // выводим заголовок
    pDC->TextOut(25, y, strHeader);

    // формируем смещение для линии
    y -= tm.tmHeight + 5;

    // определяем длину текста
    CSize len = pDC->GetTextExtent(strHeader);

    // рисуем разделительную линию
    pDC->MoveTo(0, y);
    pDC->LineTo(len.cx + 250, y);

    pDC->SetTextAlign(TA_RIGHT);
    TCHAR szBuf[32];

    // формируем строку с номером страницы
    wsprintf(szBuf, "%d", pInfo->m_nCurPage);

    // выводим номер страницы в правом нижнем углу
    pDC->TextOut(pInfo->m_rectDraw.right - 525,
                 pInfo->m_rectDraw.bottom + 1250, szBuf);
}
```

Предварительный просмотр документа

Помимо рассмотренных возможностей вывода документа на экран и на принтер, библиотека классов MFC поддерживает еще один не менее важный и интересный режим. Речь идет о предварительном просмотре документа перед печатью, т. е. получении на экране образа страницы (или страниц), в том виде, в каком она будет выведена на принтер.

Для начала перечислим те возможности, которые библиотека предоставляет "по умолчанию". Другими словами, то, что получаешь, добавив к своей программе единственную строчку кода в карту сообщений своего класса представления:[1]

```
ON_COMMAND(ID_FILE_PRINT_PREVIEW, CScrollView::OnFilePrintPreview)
```

А возможности достаточно широкие и включают, помимо отображения уменьшенного образа одной или двух страниц документа, печать изображаемых(ой) страниц(ы) или всего документа, переход к предыдущей и/или следующей странице, масштабирование изображения и выход из режима предварительного просмотра. А если предварительно сообщить библиотеке MFC размер документа, т. е. число страниц, то она дополнительно выведет полосу прокрутки для перехода от одной страницы к другой.

---

[1] Если за вас это еще не сделал AppWizard (надеемся, что не требует пояснения возможность использования вместо класса *CScrollView* любого другого класса представления).

Но... нет предела совершенству, всегда хочется чего-то большего. И библиотека классов MFC предоставляет достаточно широкие возможности для реализации собственных замыслов. Начнем с самого простого, опустив вопросы различия контекстов принтера и экрана.

Для режима предварительного просмотра характерна та же последовательность действий, что и при выводе на печать (рис. 49). Библиотека MFC сама формирует необходимый контекст устройства. Поэтому, как и при печати документа, задачей программиста является, всего лишь, переопределение функций *OnPreparePrinting*, *OnBeginPrinting*, *OnPrepareDC*, *OnPrint* и *OnEndPrinting*. Поскольку эти действия достаточно просты, ограничимся рассмотрением нескольких фрагментов кода программы (с подробными комментариями), которые иллюстрируют описываемый материал примером вывода рамки вокруг текста в режиме предварительного просмотра (рис. 51).

```
BOOL CNoteView::OnPreparePrinting(CPrintInfo* pInfo)
{
  if (pInfo != NULL && !pInfo->m_bPreview)
  {
    // подготовка данных для инициализации блока диалога
    // Print (Печать)
    ...
  }

  // не забываем вызывать функцию базового класса
  return DoPreparePrinting(pInfo);
}

void CNoteView::OnPrint(CDC* pDC, CPrintInfo* pInfo)
{
  ...

  // в режиме предварительного просмотра рисуем рамку вокруг текста
  if (pInfo != NULL && pInfo->m_bPreview)
    DrawMargins(pDC, rect);

  ...
  // выводим сам текст
  OnDraw(pDC);
}

void CNoteView::DrawMargins(CDC* pDC, CRect &rt)
{
  CNDApp *pApp = (CNDApp*)AfxGetApp();
  if (pDC->m_hAttribDC != NULL)
  {
    CRect rect, rectMargin = pApp->m_rtMargin;

    // устанавливаем границы для рамки
    rect.left   =  pApp->m_rtMargin.left - 5;
    rect.right  =  rt.right - pApp->m_rtMargin.right + 10;
```

```
    rect.top    = -pApp->m_rtMargin.top;
    rect.bottom = -rt.bottom + pApp->m_rtMargin.bottom;

    // создаем карандаш
    CPen pen(PS_DOT, 0, pDC->GetTextColor());

    // выбираем его в контекст устройства
    CPen* ppen = pDC->SelectObject(&pen);

    // рисуем рамку
    pDC->MoveTo(0, rect.top);
    pDC->LineTo(10000, rect.top);
    pDC->MoveTo(rect.left, 0);
    pDC->LineTo(rect.left, -10000);
    pDC->MoveTo(0, rect.bottom);
    pDC->LineTo(10000, rect.bottom);
    pDC->MoveTo(rect.right, 0);
    pDC->LineTo(rect.right, -10000);

    // удаляем карандаш из контекста устройства
    pDC->SelectObject(ppen);
  }
}
```

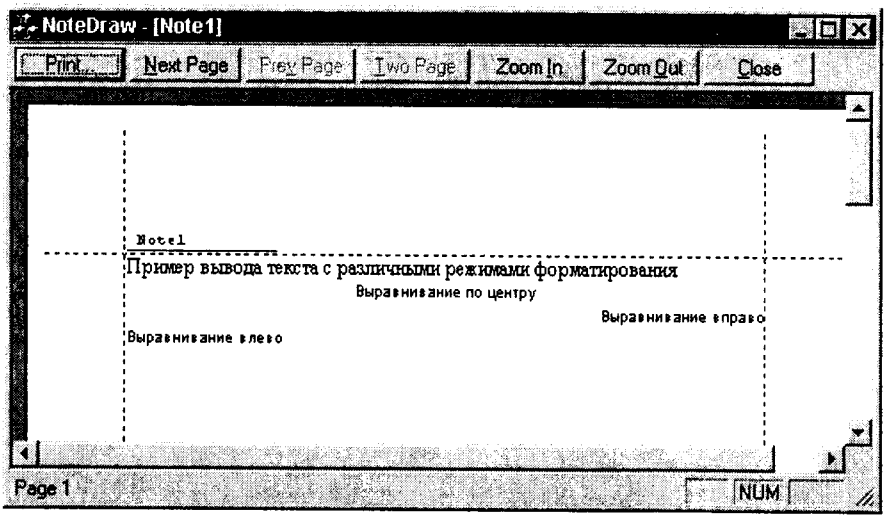


Рис. 51. Режим предварительного просмотра документа

Рассмотренной техники вполне достаточно в подавляющем большинстве случаев. Однако разработчики библиотеки MFC изменили бы себе, если бы не предоставили дополнительных возможностей для работы в этом режиме.

Итак, при выполнении команды ID_FILE_PRINT_PREVIEW вместо непосредственного отображения образа документа на принтере необходима его

эмуляция на экране. Для этого в библиотеке MFC реализованы два специальных класса: *CPreviewDC* и *CPreviewView*. Основную роль в эмуляции принтера играет объект класса *CPreviewDC*, производного от *CDC*. Как известно, все объекты *CDC* содержат члены двух контекстов устройства – собственно контекст устройства и контекст отображения, которые обычно совпадают. Но у объекта класса *CPreviewDC* они используются для разных целей: контекст устройства представляет принтер, а контекст отображения – экран, на который и осуществляется вывод. И когда приложение выполняет какие-либо операции, которые изменяют характеристики контекста принтера, библиотека MFC автоматически выполняет необходимые операции эмуляции для контекста экрана. Например, если приложение устанавливает какой-либо шрифт для печати, то библиотека выбирает шрифт для экрана, который эмулирует шрифт принтера.

Вторым классом, участвующим в организации работы режима предварительного просмотра, является *CPreviewView*, базирующийся на классе *CView*. Этот класс предоставляет поддержку для уникальных особенностей окна предварительного просмотра, таких как отображение и функционирование панели инструментов, возможность просмотра одной или двух страниц, изменение масштаба изображения, например, увеличение просматриваемого образа. Класс спроектирован достаточно органично и в большинстве случаев нет необходимости непосредственно вызывать или переопределять какую-либо его функцию. Другое дело, если требуется реализовать свой интерфейс для предварительного просмотра (например, если необходима поддержка редактирования в этом режиме). Для тех, кому это может понадобиться, рассмотрим необходимую последовательность действий.

Поскольку все начинается с выбора команды ID_FILE_PRINT_PREVIEW, которой соответствует обработчик *OnFilePrintPreview*, рассмотрим его реализацию, предоставляемую по умолчанию, поскольку при переопределении придется делать нечто подобное:

```
void CView::OnFilePrintPreview()
{
  CPrintPreviewState* pState = new CPrintPreviewState;

  if (!DoPrintPreview(AFX_IDD_PREVIEW_TOOLBAR, this,
                      RUNTIME_CLASS(CPreviewView), pState))
  {
    TRACE0("Error: DoPrintPreview failed.\n");
    AfxMessageBox(AFX_IDP_COMMAND_FAILURE);
    delete pState;
  }
}
```

Для нас интерес здесь представляют структура библиотеки MFC *CPrintPreviewState* и функция *DoPrintPreview*. Начнем со структуры:

```
struct CPrintPreviewState {
  UINT nIDMainPane;       // идентификатор скрываемой основной области
  HMENU hMenu;            // сохраненный дескриптор меню
  DWORD dwStates;         // состояние видимости панелей элементов
                          // управления
  CView *pViewActiveOld; // сохраненное представление

// указатель на функцию, позволяющую "безопасно" выйти
// из режима предварительного присмотра
  BOOL(CALLBACK *lpfnCloseProc)(CFrameWnd *pFrameWnd);
  HACCEL hAccelTable;     // сохраненная таблица акселераторов

  CPrintPreviewState();   // конструктор
};
```

Основным ее назначением является сохранение параметров активного фрейма, которые он имел перед переходом в режим предварительного просмотра. При необходимости их в этом режиме можно использовать.

Основную роль играет, конечно, функция *DoPrintPreview*:

BOOL CView::**DoPrintPreview**(
  UINT nIDResource,
  CView* pPrintView,
  CRuntimeClass* pPreviewViewClass,
  CPrintPreviewState* pState) –

выполняет основную работу для подготовки и перехода в режим предварительного просмотра. В качестве параметров она принимает: nIDResource – идентификатор ресурса панели инструментов в режиме предварительного просмотра; pPrintView – указатель на объект "представление", для которого выполняется предварительный просмотр (чаще всего используется значение this, но оно может быть и другим, если режим предварительного просмотра реализуется не для одного представления); pPreviewViewClass – указатель на структуру времени выполнения класса CPreviewView, что необходимо для динамического создания объекта этого класса во время выполнения функции; pState – указатель на структуру CPrintPreviewState (или производную от нее, если требуются какие-либо дополнительные параметры), которая должна создаваться именно здесь – функция DoPrintPreview немодальная, и структура существует до завершения работы в этом режиме.

Для того чтобы совершенно ясно представлять важность действий, выполняемых этой функцией, перечислим основные задачи, которые она решает:

- создание объекта класса *CPreviewView* – он обеспечивает поддержку уникальных особенностей окна предварительного просмотра, таких, например, как отображение и функционирование панели инструментов;
- создание панели инструментов на основе шаблона, задаваемого своим идентификатором – это может быть как стандартный (AFX_IDD_PREVIEW_TOOLBAR), так и созданный программистом блок диалога;
- создание и перевод в активное состояние окна Windows, базирующегося на объекте класса *CPreviewView*;

- установка параметров структуры *CPrintInfo*, необходимой как при печати, так при предварительном просмотре документа – в последнем случае полю *m_bPreview* присваивается значение TRUE;
- вызов рассмотренных ранее функций *OnPreparePrinting* и *OnBeginPrinting*;
- определение размеров текущего листа бумаги и вывод на экран текущей страницы.

Как видите, задач много, и если кто-то вместо предоставляемой захочет реализовать свою функцию, то не следует выпускать какую-либо из них из поля зрения. Но прежде чем это делать, стоит задаться вопросом "а зачем?" и только после получения убедительного ответа продолжать работать в этом направлении. В действительности библиотека MFC предоставляет достаточный простор для творчества и без такого рода работы, позволяя активно использовать механизм виртуальных функций. Вот и для режима предварительного просмотра она реализует две такие функции.

virtual void CFrameWnd::**OnSetPreviewMode**(
BOOL bPreview,
CPrintPreviewState *pModeStuff) –

переводит главное окно приложения в режим предварительного просмотра и обратно, в зависимости от значения параметра bPreview. Реализация по умолчанию блокирует все стандартные панели инструментов и скрывает основное меню и рабочую область главного окна приложения, т. е. переводит фрейм MDI во временный фрейм SDI. При переопределении следует вызывать аналогичную функцию базового класса для проведения необходимых инициализирующих действий. Параметр pModeStuff содержит указатель на структуру CPrintPreviewState.

virtual void CFrameWnd::**OnEndPrintPreview**(
CDC *pDC,
CPrintInfo *pInfo
POINT point,
CPreviewView *pView) –

вызывается библиотекой классов MFC, когда пользователь выходит из режима предварительного просмотра. Реализация этой функции по умолчанию вызывает функцию OnEndPrinting и восстанавливает главное окно приложения в том состоянии, которое оно имело до начала предварительного просмотра. При ее переопределении необходим вызов функции базового класса OnEndPrintPreview после всех дополнительных действий. Параметр point определяет последнюю изображаемую точку страницы в режиме предварительного просмотра, а pView – указывает на используемый объект "представление". Именно внутри этой функции следует освободить память, занимаемую структурой CPrintPreviewState, на которую указывает pState, значение которого устанавливается функцией DoPrintPreview.

Представленных функций вполне достаточно в подавляющем большинстве случаев. Переопределяя предназначенные специально для этого функции можно без лишних усилий решать такие, например, задачи:

- как ввести полосу прокрутки для доступа к страницам документа в режиме предварительного просмотра;

- как в режиме предварительного просмотра отображать страницы не с первой, а с текущей;
- как организовать различный вывод на принтер и в окно предварительного просмотра.

Естественно, список может быть значительно шире. Мы же, в заключение этого раздела приведем небольшой пример, иллюстрирующий некоторые из перечисленных возможностей.

```
void CDrawView::OnFilePrintPreview()
{
  // при переопределении функции CView::OnFilePrintPreview
  // необходимо создать собственную структуру для сохранения
  // текущего состояния фрейма
  CPrintPreviewState* pState = new CPrintPreviewState;

  // обязательный вызов, в котором задается идентификатор
  // блока диалога для отображаемой панели инструментов
  if (!DoPrintPreview(IDD_PREVIEW, this,
                      RUNTIME_CLASS(CPreviewView), pState))
  {
    // сообщение для отображения в окне отладки
    TRACE0("Ошибка: DoPrintPreview отработала неправильно.\n");

    // сообщение для отображения на экране
    AfxMessageBox(AFX_IDP_COMMAND_FAILURE);
    delete pState;
  }
}
```

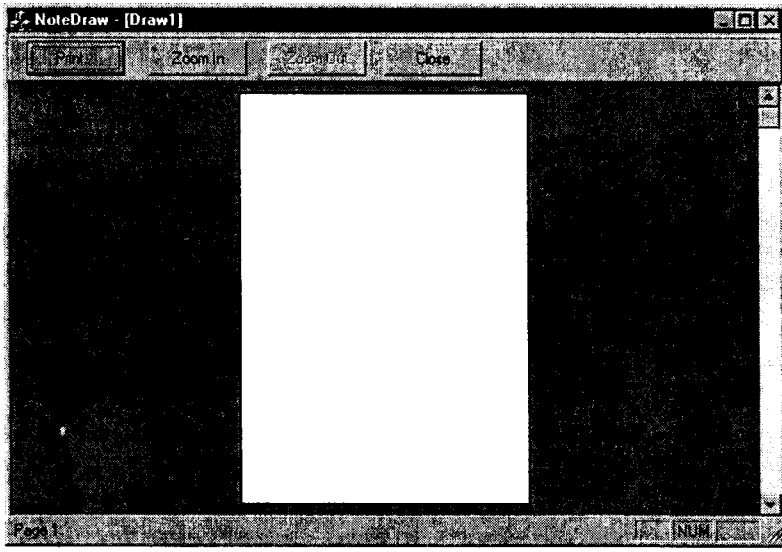


Рис. 52. Окно предварительного просмотра с измененной панелью инструментов

Внешний вид новой панели инструментов, активизирующейся в режиме предварительного просмотра, представлен на рис. 52.

Поскольку как при выводе на принтер, так и в режиме предварительного просмотра необходимо привязываться к размерам листа бумаги, то в качестве еще одного примера рассмотрим настройку блока диалога Page Setup (Параметры страницы) (рис. 53).

```
class CNoteView : public CScrollView
{
...
public:
  //{{AFX_MSG(CNoteView)
  afx_msg void OnPageSetup();
  //}}AFX_MSG
DECLARE_MESSAGE_MAP()
};

BEGIN_MESSAGE_MAP(CNoteView, CScrollView)
  //{{AFX_MSG_MAP(CNoteView)
  ON_COMMAND(ID_PAGE_SETUP, OnPageSetup)
  //}}AFX_MSG_MAP
...
END_MESSAGE_MAP()

...

void CNoteView::OnPageSetup()
{
  CNDApp *pApp = (CNDApp*)AfxGetApp();

  // создаем объект блока диалога Page Setup (Параметры страницы)
  CPageSetupDialog dlg;

  // настройка внешнего вида осуществляется с помощью структуры
  PAGESETUPDLG& psd = dlg.m_psd;

  // будем настраивать размеры полей, значения задаются
  // в сотых долях миллиметра
  psd.Flags |= PSD_MARGINS | PSD_INHUNDREDTHSOFMILLIMETERS;

  // устанавливаем необходимые значения
  psd.rtMargin.left   = pApp->m_rtMargin.left*10;
  psd.rtMargin.right  = pApp->m_rtMargin.right*10;
  psd.rtMargin.top    = pApp->m_rtMargin.top*10;
  psd.rtMargin.bottom = pApp->m_rtMargin.bottom*10;

  // для фиксации значений получаем параметры принтера
  PRINTDLG pd;
  pd.hDevNames = NULL;
  pd.hDevMode = NULL;
  pApp->GetPrinterDeviceDefaults(&pd);
```

```
  psd.hDevNames = pd.hDevNames;
  psd.hDevMode  = pd.hDevMode;

  // выводим блок диалога на экран
  if(dlg.DoModal() == IDOK)
  {
    // если пользователь нажал OK, фиксируем его установки
    pApp->m_rtMargin.left   = psd.rtMargin.left/10;
    pApp->m_rtMargin.right  = psd.rtMargin.right/10;
    pApp->m_rtMargin.top    = psd.rtMargin.top/10;
    pApp->m_rtMargin.bottom = psd.rtMargin.bottom/10;

    // при необходимости изменяем принтер,
    // используемый по умолчанию
    pApp->SelectPrinter(psd.hDevNames, psd.hDevMode, FALSE);
    CDC dc;
    pApp->CreatePrinterDC(dc);

    // получаем размеры бумаги
    pApp->m_sizePaper = CSize(dc.GetDeviceCaps(HORZSIZE)*10,
                              dc.GetDeviceCaps(VERTSIZE)*10);

    CNoteDoc* pDoc = (CNoteDoc*)GetDocument();

    // перерисовываем представления с учетом новых установок
    pDoc->UpdateAllViews(NULL, 0, this);
  }
}
```

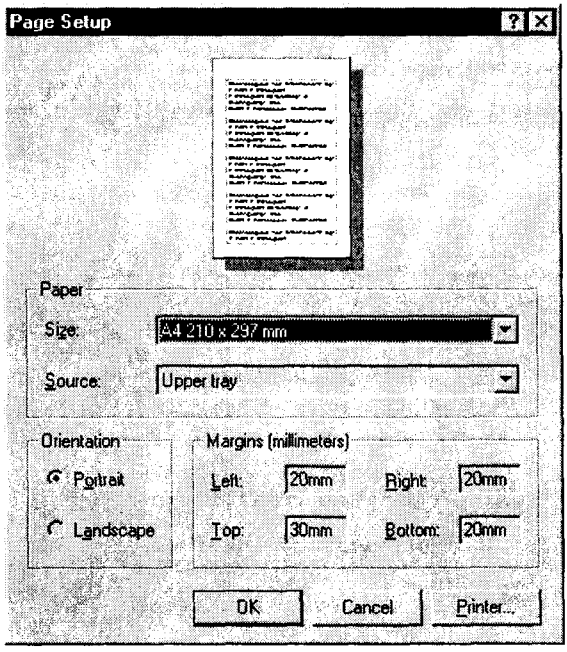


Рис. 53. Блок диалога Page Setup (Параметры страницы)

Часть XI

Элементы интерфейса пользователя

29. Меню

30. Акселераторы, графические и другие ресурсы

Глава 29

Меню

Что такое меню, как его можно создать и с ним работать? Начальные сведения об этом были приведены в главе 12, здесь мы постараемся ответить на эти вопросы более подробно. Но, прежде чем переходить к изучению средств Windows и библиотеки MFC, необходимо рассмотреть стандартные типы меню, которые можно создавать в приложениях Windows, и их основные характеристики.

Основные типы меню

Меню обычно располагается в верхней части перекрывающегося или всплывающего окна[1] под полосой заголовка, и представляют собой словесные или символические опции, дающие возможность манипулировать особенностями программы. Такое меню называется *главное меню приложения*, и содержит отдельные элементы, расположенные в *полосе меню* (menu bar), – верхний уровень иерархии (рис. 54).

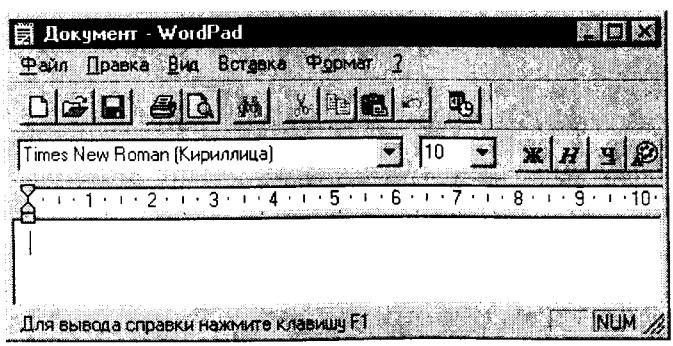


Рис. 54. Заголовок и меню приложения WordPad

[1] В общем случае дочерние окна не могут иметь полосы меню.

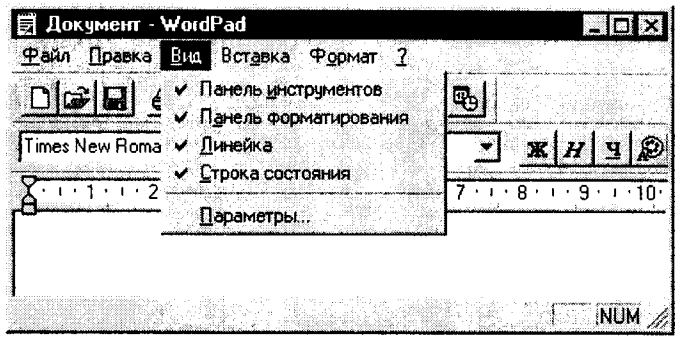


Рис. 55. Раскрытое меню

Элементы меню могут иметь различные типы и содержать ряд модификаторов (рис. 56):

- Акселератор (клавиша вызова) – сообщает пользователю, что вместо вызова меню можно вызвать команду с помощью акселератора;
- Стрелка – показывает, что при выборе данного элемента, появится подменю;
- Разделительная черта – отделяет одну группу элементов меню от другой;
- Маркер – означает, что данный параметр меню выбран;
- Многоточие – сообщает пользователю о том, что при выборе данного элемента на экран будет выведен блок диалога;
- Полутоновая надпись – говорит, что в текущем состоянии данный элемент меню недоступен;
- Знак подчеркивания – означает, что данный элемент меню можно вызвать с помощью акселератора, в который входит эта буква. Такие буквы называются мнемоническими.

Каждое окно, имеющее заголовок, может иметь *системное* меню, которое вызывается щелчком левой кнопки мыши на пиктограмме, расположенной в левой части заголовка окна (рис. 57) и представляет собой набор стандартных команд. В Windows 95 системное меню появляется, если щелкнуть правой кнопкой мыши в полосе заголовка или на пиктограмме свернутого окна.

Windows предоставляет специальный тип меню, которое можно создавать в любом месте экрана. Такое меню называется *контекстным* или *плавающим* (floating popup menu) и не привязано к полосе меню (рис. 58). Оно ассоциируется с некоторой областью окна или объектом, например, пиктограммой. Контекстное меню в некоторых случаях удобнее других, т. к. его содержимое зависит от того, для какого объекта оно было создано, оно обычно выводится на экран щелчком правой кнопки мыши.

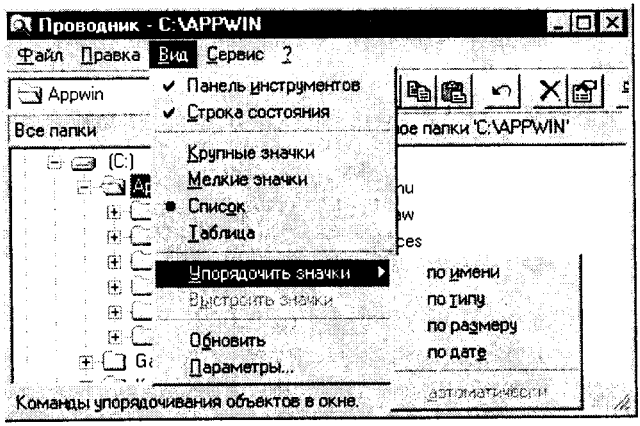


Рис. 56. Команда, имеющая подменю

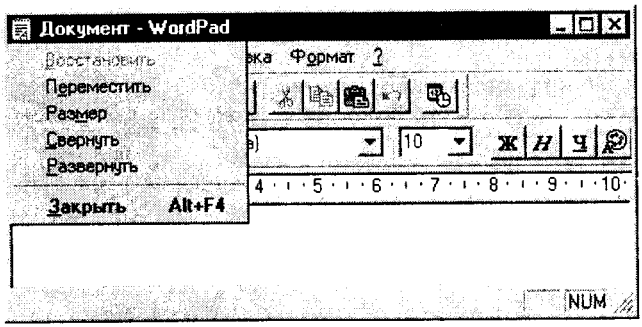


Рис. 57. Системное меню

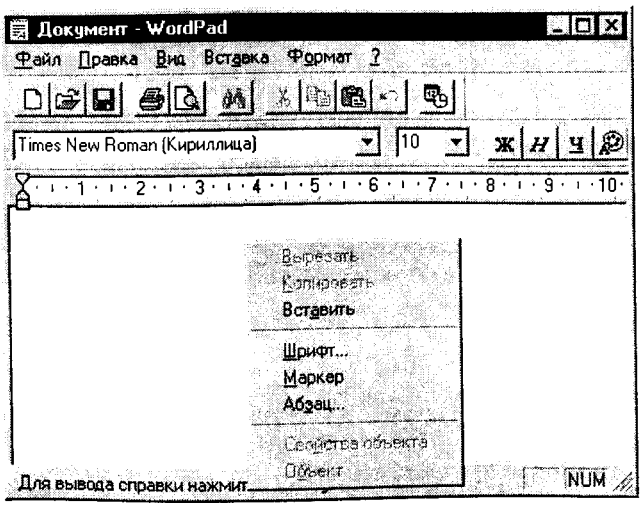
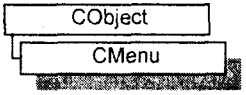


Рис. 58. Контекстное меню рабочей области окна приложения WordPad

Теперь, перечислив основные типы меню, используемые в Windows, рассмотрим те возможности, которые предоставляет библиотека MFC для работы с ними.

Точно так же, как в Win32 API есть функции, специально предназначенные для работы с меню, в библиотеке MFC есть адекватная замена этих функций, реализованная в виде класса *CMenu*, полностью вобравшего их в себя.

Класс CMenu



Это класс, который инкапсулирует объект Windows, определяемый дескриптором HMENU, и предоставляет функции для работы с меню. С помощью этого класса приложение может создавать меню, даже не имея его шаблона, добавлять или удалять элементы или подменю, управлять их состоянием — активизировать или блокировать, отмечать команды и т. д.

Рассмотрим основные члены этого класса:

CMenu::**m_hMenu** —
: определяет дескриптор HMENU меню Windows, присоединенного к объекту класса, для создания которого, как обычно, используется конструктор.

CMenu::**CMenu**() —
: конструктор. Следует совершенно четко представлять, что при создании объекта не происходит автоматического создания самого меню. Для того чтобы это реализовать, необходимо вызвать одну из функций, отвечающих за создание или загрузку меню.

Для создания "пустого" меню, т. е. меню, не содержащего ни одной строчки и ни одного подменю, достаточно воспользоваться функцией:

BOOL CMenu::**CreateMenu**() —
: создает пустое меню Windows и присоединяет его к объекту класса. Добавлять в него элементы можно, используя функции AppendMenu и InsertMenu. При успешном создании меню функция возвращает TRUE и FALSE в случае неудачи.

Для того чтобы добавить в существующее меню некоторое подменю, в классе реализована функция:

BOOL CMenu::**CreatePopupMenu**() —
: создает пустое подменю. Добавить в него элементы можно с помощью тех же функций — AppendMenu и InsertMenu. При успешном завершении функция возвращает TRUE и FALSE в случае неудачи.

Если создание меню производится на основе шаблона, то можно воспользоваться функциями

BOOL CMenu::**LoadMenu**(LPCTSTR lpszResourceName)

и

BOOL CMenu::**LoadMenu**(UINT nIDResource) —
: загружают ресурс меню из исполняемого файла приложения и присоединяют его к объекту. В качестве параметра используется либо указатель на текстовую

строку lpszResourceName, содержащую имя ресурса загружаемого меню, либо его числовой идентификатор nID.

Аналогичная функция существует для меню, создаваемого на основе шаблона в оперативной памяти:

BOOL CMenu::**LoadMenuIndirect**(const void *lpMenuTemplate) —
загружает ресурс меню из шаблона в памяти и присоединяет его к объекту CMenu. В качестве параметра принимает адрес этого шаблона, который состоит из единственного заголовка (структура *MENUITEMTEMPLATEHEADER*) и одного или нескольких наборов элементов меню (структура *MENUITEMTEMPLATE*)[1]. При успешном завершении функция возвращает TRUE, а в случае неудачи — NULL.

Рассмотрим состав указанных структур.

Структура *MENUITEMTEMPLATEHEADER* определяет заголовок шаблона меню:

```
typedef struct{
  WORD versionNumber;
  WORD offset;
} MENUITEMTEMPLATEHEADER;
```

и содержит следующие поля: *versionNumber* — должно быть равно нулю[2]; *offset* — определяет смещение (в байтах) начала списка элементов меню от конца данной структуры. Обычно его значение равно нулю, и список начинается сразу же после заголовка.

Структура *MENUITEMTEMPLATE* определяет элемент меню в шаблоне:

```
typedef struct{
  WORD mtOption;
  WORD mtID;
  WCHAR mtString[1];
} MENUITEMTEMPLATE;
```

Эта структура содержит следующие поля: *mtOption* — задает один или несколько флагов, из списка доступных,[3] для элемента меню и должно содержать MF_END для последнего элемента подменю; *mtID* — определяет идентификатор сообщения (команды) элемента меню (для подменю это поле не используется); *mtString* — определяет строку *Unicode* (с нулевым завершающим символом) с текстом элемента меню.

---

[1] Согласно документации, в Windows 95 дополнительно можно использовать расширенный шаблон, который вместо перечисленных структур использует соответственно структуры *MENUEX_TEMPLATE_HEADER* и *MENUEX_TEMPLATE_ITEM* — фактически же эти структуры не представлены ни в одном заголовочном файле, и работать с ними, естественно, нельзя.

[2] Хотя, как утверждает документация, должно определять номер версии.

[3] Список флагов приведен в разделе "Создание меню на основе шаблона".

Если по каким-либо причинам понадобилось меню, которое не присоединяется ни к одному окну, то при завершении работы с ним следует воспользоваться функцией

BOOL CMenu::**DestroyMenu**()

которая перед завершением работы приложения удаляет меню и возвращает системе занимаемые им ресурсы.

Для изменения состава меню, созданного функциями *CreateMenuIndirect* и *CreateMenu*, в классе предусмотрена специальная группа функций.

Добавление элементов в меню производится функциями *AppendMenu* и *InsertMenu*.

```
BOOL CMenu::AppendMenu(
  UINT nFlags,
  UINT nIDNewItem = 0,
  LPCTSTR lpszNewItem = NULL)
```

и

```
BOOL CMenu::AppendMenu(
  UINT nFlags,
  UINT nIDNewItem = 0,
  const CBitmap *pBmp)
```

добавляют новый элемент в конец меню. Состояние элемента задается параметром nFlags, который может содержать одно или несколько из следующих значений:

MF_CHECKED, MF_UNCHECKED	при изображении элемент меню отмечается (MF_CHECKED) или не отмечается (MF_UNCHECKED) галочкой
MF_DISABLED, MF_ENABLED	элемент меню блокирован (MF_DISABLED), но изображается обычным (черным) цветом, или доступен (MF_ENABLED)
MF_GRAYED	элемент меню блокирован и изображается серым цветом
MF_MENUBREAK	элемент меню верхнего уровня выводится с новой строки, а элемент подменю – в новом столбце без разделительной вертикальной линии
MF_MENUBARBREAK	аналогично MF_MENUBREAK, но дополнительно новый столбец отделяется вертикальной линией
MF_OWNERDRAW	меню само отвечает за изображение своего элемента
MF_POPUP	элемент меню имеет ассоциированное с ним подменю, дескриптор которого задается в параметре *nIDNewItem*
MF_SEPARATOR	элемент представляет собой горизонтальную разделительную линию; этот флаг не может комбинироваться с MF_POPUP и MF_DISABLED; при установленном данном флаге все другие игнорируются
MF_STRING	элемент представляет собой текстовую строку

При использовании этих флагов следует иметь в виду, что нельзя комбинировать следующие флаги:

- MF_DISABLED, MF_ENABLED и MF_GRAYED
- MF_STRING, MF_OWNERDRAW, MF_SEPARATOR и битовый массив
- MF_MENUBREAK и MF_MENUBARBREAK
- MF_CHECKED и MF_UNCHECKED

Параметр nIDNewItem определяет идентификатор команды нового элемента, а если nFlags установлен в MF_POPUP, то дескриптор (HMENU) подменю. Через этот параметр передается дополнительная информация о содержимом нового элемента, если nFlags принимает одно из следующих значений:

nFlags	lpszNewItem
MF_OWNERDRAW	содержит 32-битное значение, которое приложение может использовать в качестве дополнительных данных, ассоциированных с элементом меню при обработке сообщений WM_MEASUREITEM и WM_DRAWITEM
MF_SEPARATOR	значение игнорируется
MF_STRING	содержит текстовую строку

Параметр pBmp определяет указатель на объект класса CBitmap, который может использоваться в качестве содержимого элемента меню вместо текстовой строки. При этом параметр nFlags *не может* принимать значения MF_OWNERDRAW и MF_STRING.

```
BOOL CMenu::InsertMenu(
   UINT nPosition,
   UINT nFlags,
   UINT nIDNewItem = 0,
   LPCTSTR lpszNewItem = NULL)
```

и

```
BOOL CMenu::InsertMenu(
   UINT nPosition,
   UINT nFlags,
   UINT nIDNewItem = 0,
   const CBitmap *pBmp)
```

вставляют новый элемент в позицию определяемую параметром nPosition и сдвигает вниз существующие элементы. Значения всех параметров, кроме nFlags такие же, как и для функции AppendMenu. Параметр nFlags дополнительно используется для интерпретации значения nParam:

nFlags	nPosition
MF_BYCOMMAND	определяет идентификатор элемента меню, перед которым вставляется новый элемент

(продолжение)

nFlags	nPosition
MF_BYPOSITION	определяет порядковый номер элемента меню, перед которым вставляется новый элемент; если установить *nPosition* = -1, то элемент будет добавлен в конец меню

Если требуется не вставлять новый, а заменить существующий элемент меню, то можно воспользоваться функциями:

```
BOOL CMenu::ModifyMenu(
  UINT nPosition,
  UINT nFlags,
  UINT nIDNewItem,
  LPCTSTR lpszNewItem)
и
BOOL CMenu::ModifyMenu(
  UINT nPosition,
  UINT nFlags,
  UINT nIDNewItem,
  const CBitmap *pBmp)
```

Параметры этих функций идентичны параметрам функций AppendMenu и InsertMenu за исключением того, что новый элемент не вставляется в меню, а заменяет тот, на который указывает параметр nPosition.

Для удаления элемента из меню в классе предусмотрены две функции:

```
BOOL CMenu::DeleteMenu(
  UINT nPosition,
  UINT nFlags)
и
BOOL CMenu::RemoveMenu(
  UINT nPosition,
  UINT nFlags) —
```

удаляют элемент меню, задаваемый параметром nPosition, значение которого интерпретируется аналогично функции InsertMenu. Отличие между функциями заключается в том, что при удалении подменю функцией DeleteMenu все связанные с ним ресурсы освобождаются, а в случае RemoveMenu – нет, и можно снова воспользоваться удаленным элементом (по его идентификатору).

Для изменения состояния элемента меню используется функция:

```
UINT CMenu::EnableMenuItem(
  UINT nIDEnableItem,
  UINT nEnable) —
```

позволяет установить элемент меню nIDEnableItem в одно из трех состояний (задается параметром nEnable): доступное (MF_ENABLED), запрещенное (MF_DISABLED) или заблокированное (MF_GRAYED). Эти значения должны комбинироваться с одним из флагов MF_BYCOMMAND или MF_BYPOSITION, которые определяют, каким образом задается элемент – идентификатором или порядковым номером. Функция возвращает предыдущее состояние элемента или -1, если элемент не существует.

У элемента можно проставить отметку с помощью функций:

UINT CMenu::**CheckMenuItem**(
UINT nIDCheckItem,
UINT nCheck) —
устанавливает или сбрасывает галочку у элемента меню. Параметр nCheck может принимать значение MF_CHECKED или MF_UNCHECKED, как обычно в комбинации с флагами MF_BYCOMMAND или MF_BYPOSITION, которые определяют интерпретацию параметра nIDChecked – идентификатор или порядковый номер. Функция возвращает предыдущее состояние элемента или -1, если элемент не существует.

BOOL CMenu::**CheckMenuRadioItem**(
UINT nIDFirst,
UINT nIDLast,
UINT nIDItem,
UINT nFlags) —
помечает определенный элемент меню в группе переключателей, одновременно сбрасывая метку у остальных элементов группы. Для отметки элемента вместо галочки используется кружок. Параметры nIDFirst и nIDLast определяют, соответственно, первый и последний элемент в группе переключателей. Сам отмечаемый элемент задается параметром nIDItem. Параметр nFlags определяет интерпретацию этих параметров и может принимать одно из значений MF_BYCOMMAND или MF_BYPOSITION.

Вместо стандартных галочки или кружка, используемых для отметки элемента, в меню можно поместить произвольный битовый массив:

BOOL CMenu::**SetMenuItemBitmaps**(
UINT nPosition,
UINT nFlags,
const CBitmap *pBmpUnchecked,
const CBitmap *pBmpChecked) —
ассоциирует с элементом меню битовые массивы, которые будут выводиться для помеченного (pBmpChecked) и непомеченного (pBmpUnchecked) состояний. Параметры nPosition и nFlags определяют элемент меню по идентификатору или порядковому номеру. Если либо pBmpChecked, либо pBmpUnchecked равен NULL, то для соответствующего атрибута рядом с элементом ничего не отображается, а если оба параметра равны NULL, то используются стандартные галочка и кружок. При использовании этой возможности следует иметь в виду, что при разрушении меню не происходит автоматического удаления битовых массивов – эта задача возлагается на приложение.

После внесения в меню всех изменений необходимо вызвать функцию

void CWnd::**DrawMenuBar**() —
отвечает за перерисовку полосы меню.

В классе *CMenu* реализовано несколько функций, которые позволяют получить различную информацию о самом меню и о его элементах.

UINT CMenu::**GetMenuItemCount**() —
позволяет получить число элементов меню верхнего уровня и подменю.

UINT CMenu::**GetMenuItemID**(int nPos) —

возвращает идентификатор элемента подменю. Если параметр nPos определяет позицию самого подменю, то функция возвращает -1, а если позицию разделителя (элемент имеет стиль MF_SEPARATOR), возвращается 0.

UINT CMenu::**GetMenuState**(
UINT nID,
UINT nFlags) —

возвращаемое значение – состояние элемента или число элементов в выпадающем меню, определяется передаваемыми ей параметрами nID и nFlags. Параметр nID определяет элемент меню, для которого запрашивается информация и его интерпретация зависит от значения, указанного в параметре nFlags. Если там указан флаг MF_BYCOMMAND, то nID определяет идентификатор элемента, а если установлен флаг MF_BYPOSITION – порядковый номер элемента. Для выпадающего меню старший байт возвращаемого значения содержит число элементов в нем, младший байт – набор флагов, описывающих выпадающее меню. Для меню верхнего уровня возвращаемое значение является набором флагов, описывающих указанный элемент меню. Полный список флагов приведен при описании функции AppendMenu.

int CMenu::**GetMenuString**(
UINT nIDItem,
LPTSTR lpString,
int nMaxCount,
UINT nFlags)

и

int CMenu::**GetMenuString**(
UINT nIDItem,
CString &rString,
UINT nFlags) —

позволяют получить текстовую строку, соответствующую элементу меню. Возвращают число байт (без завершающего нулевого символа), скопированных в буфер. Параметры: nIDItem и nFlags действуют так же, как для всех предыдущих функций; rString – ссылка на объект класса CString, получающий строку; lpString – адрес буфера, куда должна копироваться строка; nMaxCount – размер буфера. Следует иметь в виду, что символы, не поместившиеся в буфер, будут отброшены.

CMenu* CMenu::**GetSubMenu**(int nPos) —

возвращает указатель на объект CMenu, определяющий подменю, или NULL. Если объект класса CMenu не существует, то создается временный. Параметр nPos задает позицию (начиная с 0) подменю.

Для поддержки работы с контекстными меню в классе реализована функция

BOOL CMenu::**TrackPopupMenu**(
UINT nFlags,
int x,
int y,
CWnd *pWnd,
LPCRECT lpRect = 0)

которая выводит на экран контекстное меню и создает свой собственный цикл обработки сообщений. Функция не завершается до тех пор, пока работа с меню

не будет закончена либо выбором элемента, либо отказом от выбора, после чего меню уничтожается. Параметр nFlags определяет расположение контекстного меню (относительно экрана) и кнопку мыши, с помощью которой должен выполняться выбор. Он может содержать одно из трех значений (для расположения на экране) в комбинации с одним из двух флагов для кнопки мыши

Значения параметра nFlags для расположения на экране

TMP_CENTERALIGN	центрирование относительно координаты, заданной параметром *x*
TMP_LEFTALIGN	выравнивание по левой границе относительно координаты, заданной параметром *x*
TMP_RIGHTALIGN	выравнивание по правой границе относительно координаты, заданной параметром *x*

Значения параметра nFlags для кнопки мыши

TMP_LEFTBUTTON	использование левой кнопки мыши
TMP_RIGHTBUTTON	использование правой кнопки мыши

Параметр pWnd задает окно, которое получит сообщение WM_COMMAND после того, как пользователь сделает выбор в контекстном меню, а lpRect является указателем на структуру типа *RECT* (или класса CRect), определяющую координаты прямоугольной области, в которой пользователь может выполнять выбор из меню. Если щелчок мышью будет сделан вне этой области, контекстное меню исчезнет с экрана, что эквивалентно отказу от выбора. Если для него задать значение NULL, то размеры и расположение этой прямоугольной области будут совпадать с размерами контекстного меню.

В классе представлены две виртуальные функции, которые произвольным образом позволяют изменять внешний вид меню:[1]

virtual void CMenu::**MeasureItem**(LPMEASUREITEMSTRUCT lpMeasureItemStruct) — вызывается библиотекой MFC, когда создается меню, имеющее стиль самоотображения (owner-draw). В качестве параметра в функцию передается указатель на структуру *MEASUREITEMSTRUCT*, с помощью которой можно настроить размеры меню.

virtual void CMenu::**DrawItem**(LPDRAWITEMSTRUCT lpDrawItemStruct) — вызывается библиотекой MFC, когда создается меню, имеющее стиль самоотображения. В качестве параметра в функцию передается указатель на структуру *DRAWITEMSTRUCT*, которая содержит информацию для рисования элемента меню.

При использовании стиля самоотображения обе функции должны быть обязательно переопределены, т. к. их реализация в классе *CMenu* не содержит кода.

Две оставшиеся функции класса предназначены для работы со справочной системой:

---

[1] Процесс самоотображения (owner-draw) и используемые структуры подробно описаны в главе 21.

DWORD CMenu::**GetMenuContextHelpId**()

и

BOOL CMenu::**SetMenuContextHelpId**(DWORD dwContextHelpId) —
позволяют получить или установить идентификатор контекстной справки, ассоциированной с меню.

Теперь, после теоретического знакомства с возможностями, заложенными в классе *CMenu*, можно переходить к практическим аспектам и тонкостям программирования при построении и работе с меню.

В системе Windows реализованы три способа создания меню:

1. На основе шаблона, задаваемого в файле ресурса;
2. Динамическое создание при помощи специальных функций;
3. На основе шаблона в оперативной памяти.

Рассмотрим каждый из этих способов более подробно.

Создание меню на основе шаблона

В главе 12 был приведен код из файла ресурсов, который создавал меню для программы MDI. Не будем останавливаться на этом вопросе более подробно, а приведем только описание атрибутов, определяющих внешний вид и поведение строки меню (рис. 59).

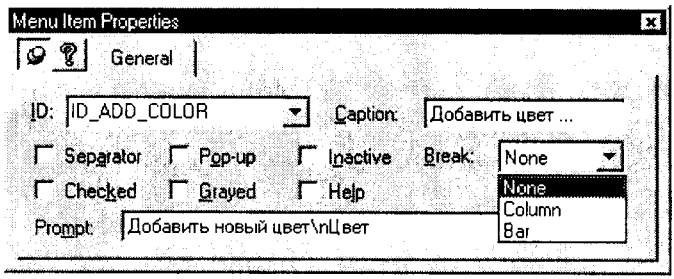


Рис. 59. Атрибуты меню в окне свойств, открытом в режиме настройки параметров меню

Атрибут	Описание
Separator	элемент представляет собой горизонтальную разделительную линию, служащую для разбиения элементов меню на группы
Checked	при выводе на экран элемент меню отмечается слева галочкой
Pop-up	элемент определяет подменю
Grayed	элемент отображается серым цветом и находится в заблокированном состоянии; несовместим с атрибутом Inactive
Help	элемент выравнивается по правому краю полосы меню

(продолжение)

Атрибут	Описание
Break	этот атрибут может принимать одно из трех значений: **None** – обычный элемент меню; **Column** – для меню верхнего уровня элемент выводится с новой строки, а для подменю – в новом столбце; **Bar** – дополнительный столбец подменю отделяется вертикальной линией.
ID	идентификатор элемента меню
Caption	текст, определяющий имя элемента меню; текстовая строка может содержать символы: **&** — буква, перед которой стоит этот знак будет подчеркнута; **\t** – включает в строку символ табуляции; **\a** – выравнивает текст по правой границе меню
Prompt	текст, который будет отображаться в строке состояния и (после \n) в окне всплывающей подсказки
Inactive	элемент находится в заблокированном состоянии, но отображается обычным цветом

После того как создан шаблон (одним из способов, описанных в главе 12), необходимо подключить меню к окну приложения. Этот вопрос также достаточно подробно рассматривался, поэтому только перечислим те места, где это можно сделать:

- При регистрации класса окна – когда меню определяется для нескольких окон одного класса:

```
wc.lpszMenuName = "IDR_MAINFRAME";
```

- При создании окна – если для него необходимо меню, отличное от указанного в классе окна:

```
if (!pMainFrame->LoadFrame(IDR_MAINFRAME))
    return FALSE;
```

- При работе в рамках архитектуры "документ/представление" – достаточно, чтобы идентификатор меню совпадал с идентификаторами фрейма, документа и представления. Тогда вопросами подключения меню полностью занимается библиотека классов MFC при создании шаблона документа:

```
pDocTemplate = new CMultiDocTemplate(
                        IDR_NOTETYPE,
                        RUNTIME_CLASS(CNoteDoc),
                        RUNTIME_CLASS(CNoteFrame),
                        RUNTIME_CLASS(CNoteView));
```

При использовании любого из перечисленных способов подключения меню к окну нет необходимости уничтожать меню перед завершением работы приложения, т. к. библиотека MFC делает это сама.

Меню, созданное в системе Windows, где взаимодействие между объектами осуществляется при помощи сообщений, также работает по этому принципу. Наиболее важным из сообщений, которое передает меню, является WM_COMMAND, означающее, что выбран тот или иной элемент меню. Ниже приведены этапы работы с меню и возникающие при этом сообщения.

Этап работы	Сообщение	Обработчик в MFC
Инициализация меню	WM_INITMENU	OnInitMenu
Вывод всплывающего меню на экран	WM_INITMENUPOPUP	OnInitMenuPopup
Инициализация и вывод на экран всплывающего меню	WM_INITMENU и WM_INITMENUPOPUP	OnInitMenu и OnInitMenuPopup
Нахождение нужного элемента меню	WM_MENUSELECT	OnMenuSelect
Выбор элемента меню	WM_COMMAND	OnCommand
Выбор элемента системного меню	WM_SYSCOMMAND	OnSysCommand

Рассмотрим, кратко, роль перечисленных сообщений.[1]

Когда пользователь активизирует элемент системного меню, сначала вызывается обработчик сообщения WM_SYSCOMMAND:

```
void CWnd::OnSysCommand(
   UINT nID,
   LPARAM lParam)
```

Первый параметр *nID* может содержать идентификатор строки системного меню, а также некоторые другие значения:

SC_CLOSE	Удаление окна
SC_HOTKEY	Активизация окна, связанного с акселератором, определенным приложением
SC_HSCROLL	Прокрутка по горизонтали
SC_KEYMENU	Выбор элемента меню при помощи акселератора
SC_MAXIMIZE или **SC_ZOOM**	Разворачивание окна
SC_MINIMIZE или **SC_ICON**	Сворачивание окна

[1] Действия приведенных обработчиков справедливы и для двух других способов создания меню.

SC_MOUSEMOVE	Выбор произведен при помощи мыши
SC_MOVE	Перемещение окна
SC_NEXTWINDOW	Переключение на следующее окно
SC_PREVWINDOW	Переключение на предыдущее окно
SC_RESTORE	Восстановление нормального положения и размеров окна
SC_SCREENSAVE	Запуск приложения-заставки (screen-saver application), определенного в разделе [boot] файла system.ini
SC_SIZE	Изменение размеров окна
SC_TASKLIST	Запуск или активизация приложения Task Manager
SC_VSCROLL	Прокрутка по вертикали

Кроме этих, можно использовать и свои идентификаторы, как показано в рассматриваемом ниже примере.

Второй параметр *lParam* используется только для значений первого параметра:

nID	**Значение параметра**
SC_HOTKEY	содержит идентификатор активизируемого окна
SC_MOUSEMOVE	младшее слово содержит *x*-координату, а старшее – *y*-координату

Перед отображением всплывающего меню (один раз, при активизации), Windows посылает сообщение WM_INITMENU с тем, чтобы приложение могло изменить меню перед тем, как его увидит пользователь. При этом библиотека MFC вызывает обработчик:

void CWnd::**OnInitMenu**(CMenu *pMenu)

Параметр pMenu задает указатель на объект CMenu, определяющий активизируемое меню. Обработка может заключаться в активизации или деактивизации строк меню, изменении их состояния и т. п.

Когда пользователь выбирает элемент, имеющий подменю, Windows посылает владельцу сообщение WM_INITMENUPOPUP перед отображением всплывающего меню. Библиотека MFC вызывает обработчик, который дает приложению возможность изменить всплывающее меню перед его отображением:

```
void CWnd::OnInitMenuPopup(
  CMenu pPopupMenu,
  UINT nIndex,
  BOOL bSubMenu)
```

pPopupMenu – указатель на объект "всплывающее меню"; nID – индекс всплывающего меню в меню верхнего уровня; bSubMenu – определяет, является ли всплывающее меню системным (TRUE) или нет.

Каждый раз пользователь перемещается от одного элемента меню к другому, Windows посылает сообщение WM_MENUSELECT, которое идентифициру-

ет текущий элемент, на что библиотека MFC реагирует вызовом обработчика

```
void CWnd::OnMenuSelect(
  UINT nItemID,
  UINT nFlags,
  HMENU hSubMenu)
```

nItemID – идентификатор выделенного элемента или индекс всплывающего меню; nFlags — содержит комбинацию следующих флагов, определяющих атрибуты элемента меню:

MF_BITMAP	элемент является битовым массивом
MF_CHECKED	элемент отмечается галочкой
MF_DISABLED	элемент является заблокированным, но отображается в нормальном виде
MF_GRAYED	элемент отображается серым цветом и не может быть выбран
MF_MOUSESELECT	элемент выделен мышью
MF_OWNERDRAW	элемент рисуется окном, создавшим меню
MF_POPUP	с данным элементом связывается всплывающее меню
MF_SEPARATOR	элемент является разделителем
MF_SYSMENU	элемент принадлежит системному меню

Последний параметр hSubMenu используется в двух случаях, зависящих от состояния параметра nFlags:

nFlags	Значение параметра
MF_SYSMENU	идентифицирует меню, ассоциированное с сообщением
MF_POPUP	идентифицирует дескриптор меню верхнего уровня

Многие приложения используют это сообщение для отображения дополнительной информации о выбранном элементе меню в строке состояния.

Случай, когда пользователь выбирает элемент меню, а Windows посылает сообщение WM_COMMAND, уже не раз описывался в предыдущих примерах — необходимо либо воспользоваться одним из предопределенных обработчиков стандартных команд меню, либо создать свой, например, так как было показано в главе 12.

В Windows 95 появилось еще одно сообщение, предназначенное для работы с контекстными меню – WM_CONTEXTMENU.

На наш взгляд нет необходимости дольше задерживаться на этом вопросе, поскольку при использовании шаблонов подавляющую часть работы выполняет операционная система. И все необходимые действия для работы с меню представляют относительную сложность только при построении самого первого меню.

Динамическое создание меню (приложение DynMenu)

В приведенном ниже примере показаны основные принципы работы с меню "в динамическом режиме". В нем использовано большинство функций класса *CMenu*, предназначенного, в частности, для динамического создания и изменения состава меню.

За основу взято SDI-приложение, построенное с использованием мастера AppWizard, поэтому мы не будем приводить весь код, а ограничимся фрагментами, которые иллюстрируют соответствующие аспекты работы с меню в динамическом режиме. Сразу после запуска приложения в полосе меню определены названия стандартных раскрывающихся меню, доступные для выбора, одно меню заблокировано, его название изображается серым цветом (рис. 60). Для того чтобы открыть доступ к нему, необходимо в системном меню выбрать элемент "Разблокировать меню Цвет" (рис. 61), который при этом будет отмечен галочкой.

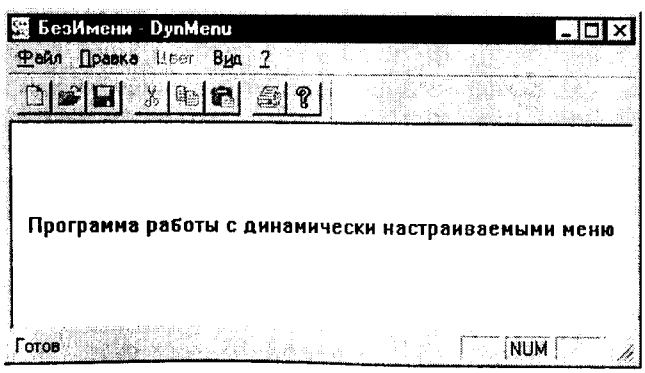


Рис. 60. Главное окно приложения DynMenu

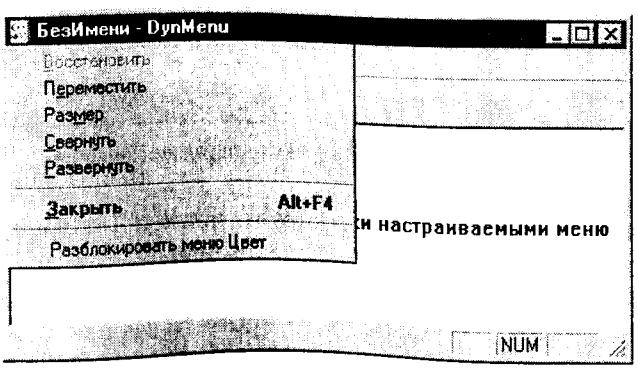


Рис. 61. Системное меню приложения DynMenu

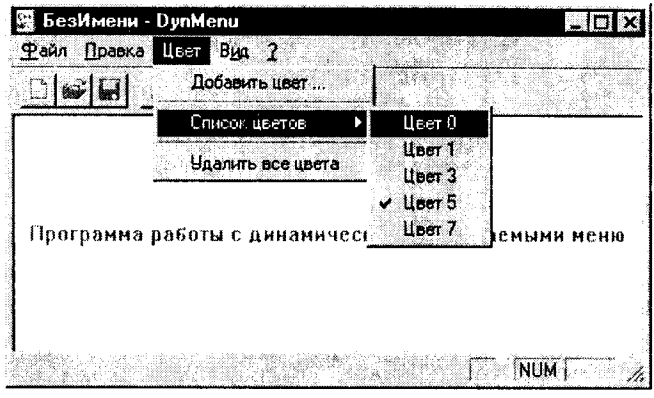


Рис. 62. Список выбранных и доступных цветов приложения DynMenu

После этого можно выбирать любой цвет из первой строки списка цветов стандартного блока диалога Color (Цвет), который выводится на экран по команде "Добавить цвет". Соответствующий цвет будет добавлен к новому, формируемому подменю команды "Список цветов" (рис. 62).

Удалить цвета из этого списка можно двумя способами: выбрав элемент меню Цвет "Удалить все цвета" для очистки списка или с помощью контекстного меню, появляющегося на экране при щелчке правой кнопкой мыши в окне приложения (рис. 63) – выбранный в нем цвет будет удален из списка.

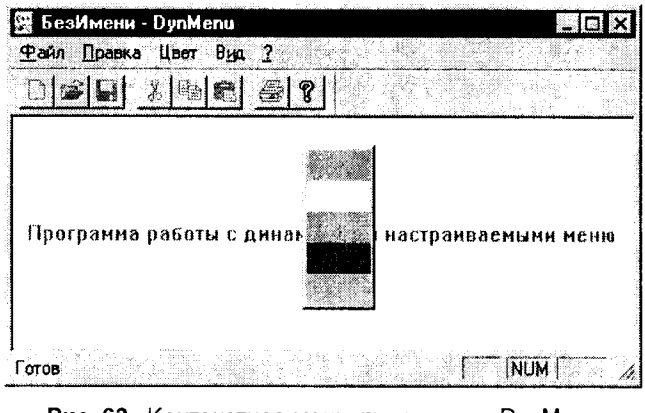


Рис. 63. Контекстное меню приложения DynMenu

Кроме этого, приложение демонстрирует пример создания собственных маркеров, используемых вместо стандартной галочки (рис. 64).

Теперь, после описания возможностей приложения, переходим непосредственно к фрагментам кода, разбив их на соответствующие темы. Фрагменты снабжены подробными комментариями и не требуют дополнительных пояснений.

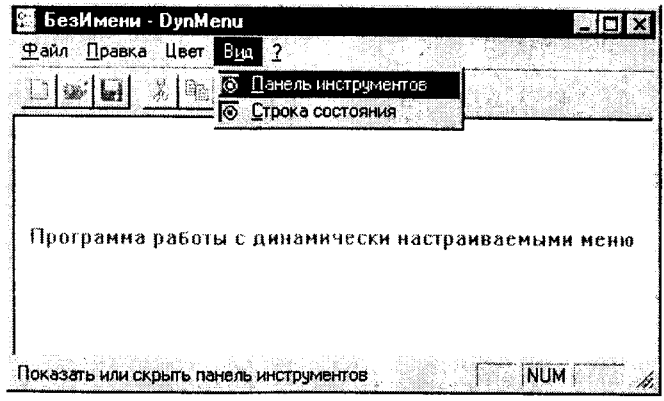


Рис. 64. Новый вид маркеров перед элементами меню в приложении DynMenu

Динамическое изменение меню

```
/*************************************************************
Файл MainFrm.h
Copyright (c) 1997 Мешков А., Тихомиров Ю., Шведов Д.
*************************************************************/
class CMainFrame : public CFrameWnd
{
protected: // create from serialization only
  CMainFrame();
  DECLARE_DYNCREATE(CMainFrame)

// Attributes
public:
  ...
// объект для создания динамического раскрывающегося меню
  CMenu   m_PopupMenu;

// Operations
public:

// функция, изменяющая состав динамического раскрывающегося меню
  void ChangeColorMenu();

...

// Generated message map functions
protected:
  //{{AFX_MSG(CMainFrame)

    ...
    // Обработчик команды "Удалить все цвета"
    afx_msg void OnDeleteAll();
```

```
  //}}AFX_MSG
  DECLARE_MESSAGE_MAP()
};

/*************************************************************
Файл MainFrm.cpp
Copyright (c) 1997 Мешков А., Тихомиров Ю., Шведов Д.
*************************************************************/
IMPLEMENT_DYNCREATE(CMainFrame, CFrameWnd)

BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
  //{{AFX_MSG_MAP(CMainFrame)
    ...
    ON_COMMAND(ID_DELETE_ALL, OnDeleteAll)
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()

void CMainFrame::ChangeColorMenu()
{
  // получаем доступ к документу
  CDynMenuDoc* pDoc = (CDynMenuDoc*)GetActiveDocument();
  ASSERT_KINDOF(CDynMenuDoc, pDoc);

// создаем два указателя на класс CMenu для доступа
// к меню верхнего уровня pTopMenu и динамическому меню цветов
  CMenu* pColorMenu = NULL;
  CMenu* pTopMenu = AfxGetMainWnd()->GetMenu();
  int iPos;

  // ищем элемент меню "Добавить цвет"
  // по идентификатору ID_ADD_COLOR
  for (iPos = pTopMenu->GetMenuItemCount()-1; iPos >= 0; iPos--)
  {
   // получаем указатель на элемент меню верхнего уровня
    CMenu* pMenu = pTopMenu->GetSubMenu(iPos);
    if (pMenu && pMenu->GetMenuItemID(0) == ID_ADD_COLOR)
    {
     // если элемент - "Цвет", то прекращаем поиск
      pColorMenu = pMenu;
      break;
    }
  }
// поиск должен закончиться успешно, иначе - ошибка
  ASSERT(pColorMenu != NULL);

  // обновляем состав подменю команды "Список цветов"

  // сначала удаляем все элементы подменю...
  for (iPos = pColorMenu->GetMenuItemCount()-3; iPos > 0; iPos--)
    pColorMenu->DeleteMenu(iPos, MF_BYPOSITION);
  m_PopupMenu.m_hMenu = NULL;
```

```
  // затем вставляем все выбранные цвета и полосу разделителя
  if(pDoc->m_icolor != -1)
  {
    // создаем подменю
    m_PopupMenu.CreatePopupMenu();

    // и вставляем его между разделителем
    // и элементом "Удалить все цвета"
    pColorMenu->InsertMenu(ID_DELETE_ALL,
                           MF_STRING | MF_POPUP | MF_BYCOMMAND,
                           (UINT)m_PopupMenu.m_hMenu, "Список цветов");
  }
    // вставляем элементы подменю
  for (int i = 0; i < NUM_COLOR; i++)
  {
    if (pDoc->m_aColor[i].m_btr == TRUE)
    {
      TCHAR strColor[64];
      sprintf(strColor, "Цвет %d", i);
      m_PopupMenu.AppendMenu(MF_STRING,
                             pDoc->m_aColor[i].m_nID, strColor);
    }
  }
  // вставляем разделитель
  if(pDoc->m_icolor != -1)
  pColorMenu->InsertMenu(ID_DELETE_ALL,
                         MF_BYCOMMAND | MF_SEPARATOR);
}

void CMainFrame::OnDeleteAll()
{
  // получаем доступ к документу
  CDynMenuDoc* pDoc = (CDynMenuDoc*)GetActiveDocument();
  ASSERT_KINDOF(CDynMenuDoc, pDoc);

  // сбрасываем маркер элемента
  for (int i = 0; i < NUM_COLOR; i++)
  {
    pDoc->m_aColor[i].m_btr = FALSE;
  }
  // определяем, что нет доступного текущего цвета
  pDoc->m_icolor = -1;
  // обновляем содержимое меню
  ChangeColorMenu();
  // обновляем изображение в окне документа
  pDoc->UpdateAllViews(NULL);
}
```

```
/*************************************************************
Файл DynMenuDoc.h
Copyright (c) 1997 Мешков А., Тихомиров Ю., Шведов Д.
*************************************************************/
// максимальное разрешенное число элементов подменю
#define NUM_COLOR 8

struct MenuColor
{
  int      m_pos; // порядковый номер в меню
  UINT     m_nID; // идентификатор меню
  COLORREF m_clr; // цвет
  BOOL     m_btr; // включать ли в список
};

class CDynMenuDoc : public CDocument
{
protected: // create from serialization only
  CDynMenuDoc();
  DECLARE_DYNCREATE(CDynMenuDoc)

// Attributes
public:
  BOOL bDelete;       // флаг удаления или вставки элемента в меню,
  int       m_icolor; // текущий цвет,
  static MenuColor m_aColor[NUM_COLOR]; // массив элементов меню

// Operations
public:
  void DoUpdateSelectColor(int iColor, CCmdUI*pCmdUI);
  void DoSelectColor(int iColor);
...

// Generated message map functions
protected:
  //{{AFX_MSG(CDynMenuDoc)

    // обработчик команды ID_COLOR для добавления элемента в меню
    afx_msg void OnAddColor();
  //}}AFX_MSG
  DECLARE_MESSAGE_MAP()
};

/*************************************************************
Файл DynMenuDoc.cpp
Copyright (c) 1997 Мешков А., Тихомиров Ю., Шведов Д.
*************************************************************/
// инициализация массива цветов: исходно все недоступны
MenuColor CDynMenuDoc::m_aColor[NUM_COLOR] =
{
  {0, ID_COLOR0,  8421631, FALSE},
  {0, ID_COLOR1,  8454143, FALSE},
```

```
  {0, ID_COLOR2,  8454016, FALSE},
  {0, ID_COLOR3,  8453888, FALSE},
  {0, ID_COLOR4, 16777088, FALSE},
  {0, ID_COLOR5, 16744448, FALSE},
  {0, ID_COLOR6, 12615935, FALSE},
  {0, ID_COLOR7, 16744703, FALSE}
};

IMPLEMENT_DYNCREATE(CDynMenuDoc, CDocument)

BEGIN_MESSAGE_MAP(CDynMenuDoc, CDocument)
  //{{AFX_MSG_MAP(CDynMenuDoc)
    ON_COMMAND(ID_ADD_COLOR, OnAddColor)
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()

//////////////////////////////////////////////////////
// CDynMenuDoc construction/destruction

CDynMenuDoc::CDynMenuDoc()
{
  m_icolor = -1;
  bDelete  = FALSE;
}

void CDynMenuDoc::OnAddColor()
{
  // настраиваем стандартный блок диалога Color (Цвет)
  CColorDialog dlg;
  dlg.m_cc.Flags |= CC_PREVENTFULLOPEN;

  if(dlg.DoModal() == IDOK)
  {
    // получаем выбранный цвет
    COLORREF color = dlg.GetColor();

    // если его еще нет в списке доступных цветов,
    // то добавляем его туда
    for(int i = 0; i < NUM_COLOR; i++)
    {
      if(color == m_aColor[i].m_clr)
      {
         m_aColor[i].m_btr = TRUE;
         m_icolor = i;
         break;
      }
    }

    // получаем доступ к текущему представлению
    POSITION pos = GetFirstViewPosition();
    CView* pview = GetNextView(pos);
    ASSERT(pview != NULL);
```

```
    // получаем доступ к активному фрейму
    CMainFrame* pwnd = (CMainFrame*)pview->GetParentFrame();
    ASSERT_KINDOF(CMainFrame, pwnd);

    // обновляем состав доступных цветов
    pwnd->ChangeColorMenu();

    // обновляем изображение
    UpdateAllViews(NULL);
  }
}

void CDynMenuDoc::DoSelectColor(int iColor)
{
  ASSERT(iColor >= 0 && iColor < NUM_COLOR);
  ASSERT(m_aColor[iColor].m_btr == TRUE);

  m_icolor = iColor;
  UpdateAllViews(NULL);
}

void CDynMenuDoc::DoUpdateSelectColor(int iColor, CCmdUI*pCmdUI)
{
  ASSERT(iColor >= 0 && iColor < NUM_COLOR);
  ASSERT(m_aColor[iColor].m_btr == TRUE);

  pCmdUI->SetCheck(m_icolor == iColor);
  pCmdUI->Enable(TRUE);
}

// поскольку используемые команды не зарегистрированы,
// внедряемся в стандартный маршрут команд для перехвата
BOOL CDynMenuView::OnCmdMsg(UINT nID, int nCode, void* pExtra,
                            AFX_CMDHANDLERINFO* pHandlerInfo)
{
  CDynMenuDoc* pDoc = GetDocument();
  ASSERT_VALID(pDoc);

  // If pHandlerInfo is NULL, then handle the message
  if (pHandlerInfo == NULL)
  {
    // организуем фильтр для перехвата нужной команды
    for (int i = NUM_COLOR-1; i >= 0; i--)
    {
      if (nID == pDoc->m_aColor[i].m_nID)
      {
        if (nCode == CN_COMMAND)
        {
          // обрабатываем перехваченную команду

          // отдельная обработка для удаления элемента и его вставки
          if(pDoc->bDelete)
```

```
            {
              pDoc->bDelete = FALSE;
              pDoc->m_aColor[i].m_btr = FALSE;
            }
            else
              pDoc->DoSelectColor(i);

            ...
          }
          else if (nCode == CN_UPDATE_COMMAND_UI)
          {
            // получаем доступ к фрейму
            pDoc->DoUpdateSelectColor(i, (CCmdUI*)pExtra);
          }
          return TRUE;
        }
      }
    }

    // не забываем вернуть управление в стандартный маршрут команд
    return CView::OnCmdMsg(nID, nCode, pExtra, pHandlerInfo);
}
```

Системное меню

```
/////////////////////////////////////////////////////////////////////////
Файл MainFrm.h
Copyright (c) 1997 Мешков А., Тихомиров Ю., Шведов Д.
/////////////////////////////////////////////////////////////////////////

class CMainFrame : public CFrameWnd
{
protected: // create from serialization only
  CMainFrame();
  DECLARE_DYNCREATE(CMainFrame)

  ...

  // ClassWizard generated virtual function overrides
  //{{AFX_VIRTUAL(CMainFrame)
  public:

    ...

    // переопределяем обработчик системных команд
    afx_msg void OnSysCommand( UINT nID, LPARAM lParam);
  //}}AFX_VIRTUAL

  DECLARE_MESSAGE_MAP()
};
```

```
/*************************************************************
Файл MainFrm.cpp
Copyright (c) 1997 Мешков А., Тихомиров Ю., Шведов Д.
*************************************************************/

IMPLEMENT_DYNCREATE(CMainFrame, CFrameWnd)

BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
  //{{AFX_MSG_MAP(CMainFrame)
    ON_WM_SYSCOMMAND()

    ...

  //}}AFX_MSG_MAP
END_MESSAGE_MAP()

void CMainFrame::ActivateFrame(int nCmdShow)
{

...

  CMenu *pViewMenu = NULL;
  CMenu *pTopMenu = AfxGetMainWnd()->GetMenu();
  int iPos;

// добавляем элемент "Разблокировать меню Цвет" в системное меню

// IDM_COLOR должен находиться в границах системных команд
  ASSERT((IDM_COLOR & 0xFFF0) == IDM_COLOR);
  ASSERT(IDM_COLOR < 0xF000);

  // получаем доступ к системному меню
  CMenu* pSysMenu = GetSystemMenu(FALSE);
  CString strColorMenu("Разблокировать меню Цвет");

  // добавляем в него разделитель и новый элемент
  // "Разблокировать меню Цвет"
  pSysMenu->AppendMenu(MF_SEPARATOR);
  pSysMenu->AppendMenu(MF_BYCOMMAND | MF_STRING,
                       IDM_COLOR, strColorMenu);

  // сбрасываем метку, показывая, что элемент "Цвет"
  // меню приложения заблокирован
  pSysMenu->CheckMenuItem(IDM_COLOR, MF_BYCOMMAND | MF_UNCHECKED);

  // ищем элемент меню Add Color по идентификатору ID_ADD_COLOR
  for (iPos = pTopMenu->GetMenuItemCount()-1; iPos >= 0; iPos--)
  {
    CMenu* pMenu = pTopMenu->GetSubMenu(iPos);
    if (pMenu && pMenu->GetMenuItemID(0) == ID_ADD_COLOR)
    {
      break;
    }
  }
```

```
  // разблокируем элемент "Цвет" в полосе меню
  pTopMenu->EnableMenuItem(iPos,
                           MF_BYPOSITION | MF_GRAYED);
  // перерисовываем полосу меню
  DrawMenuBar();

  // передаем управление фрейму
  CFrameWnd::ActivateFrame(nCmdShow);
}

// поскольку команда не зарегистрирована,
// необходимо переопределить обработчик
void CMainFrame::OnSysCommand( UINT nID, LPARAM lParam)
{
  if ((nID & 0xFFF0) == IDM_COLOR)
  {
    CMenu* pColorMenu = NULL;
    CMenu* pTopMenu = AfxGetMainWnd()->GetMenu();
    int iPos;

    // ищем элемент меню "Добавить цвет" по идентификатору
    // ID_ADD_COLOR
    for (iPos = pTopMenu->GetMenuItemCount()-1; iPos >= 0; iPos--)
    {
      CMenu* pMenu = pTopMenu->GetSubMenu(iPos);
      if (pMenu && pMenu->GetMenuItemID(0) == ID_ADD_COLOR)
      {
        break;
      }
    }

    // получаем доступ к системному меню
    CMenu *pSysMenu = GetSystemMenu(FALSE);

    // определяем текущее состояние элемента
    UINT nState = pSysMenu->GetMenuState(IDM_COLOR, MF_BYCOMMAND);

    // если элемент отмечен, то убираем маркер и запрещаем
    // доступ к элементу "Цвет" полосы меню, и наоборот
    if(nState == MF_CHECKED)
    {
      pSysMenu->CheckMenuItem(IDM_COLOR,
                              MF_BYCOMMAND | MF_UNCHECKED);
      pTopMenu->EnableMenuItem(iPos,
                               MF_BYPOSITION | MF_GRAYED);
    }
    else
    {
      pSysMenu->CheckMenuItem(IDM_COLOR,
        MF_BYCOMMAND | MF_CHECKED);
```

```
      pTopMenu->EnableMenuItem(iPos,
        MF_BYPOSITION | MF_ENABLED);
    }

    // перерисовываем полосу меню
    DrawMenuBar();
  }
  else
  {
    // если команда не наша, необходимо
    // передать ее для корректной обработки
    CWnd::OnSysCommand(nID, lParam);
  }
}
```

Контекстное меню

```
/*************************************************************
Файл DynMenuView.h
Copyright (c) 1997 Мешков А., Тихомиров Ю., Шведов Д.
*************************************************************/

class CDynMenuView : public CView
{
protected: // create from serialization only
  CDynMenuView();
  DECLARE_DYNCREATE(CDynMenuView)

// Attributes
public:
  CDynMenuDoc* GetDocument();

// Overrides
// ClassWizard generated virtual function overrides
  //{{AFX_VIRTUAL(CDynMenuView)
  public:

    ...
    // функция для внедрения в стандартный маршрут команд
    // добавления и удаления элементов меню
    virtual BOOL OnCmdMsg(UINT nID, int nCode, void* pExtra,
                          AFX_CMDHANDLERINFO* pHandlerInfo);
  //}}AFX_VIRTUAL

// Generated message map functions
  protected:
  //{{AFX_MSG(CDynMenuView)

    // обработчик для создания контекстного меню
    afx_msg void OnRButtonDown(UINT nFlags, CPoint point);
```

```
  //}}AFX_MSG
  DECLARE_MESSAGE_MAP()
};

/*************************************************************
Файл DynMenuView.cpp
Copyright (c) 1997 Мешков А., Тихомиров Ю., Шведов Д.
*************************************************************/

IMPLEMENT_DYNCREATE(CDynMenuView, CView)

BEGIN_MESSAGE_MAP(CDynMenuView, CView)
  //{{AFX_MSG_MAP(CDynMenuView)
    ON_WM_RBUTTONDOWN()
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()

void CDynMenuView::OnRButtonDown(UINT nFlags, CPoint point)
{
  CDynMenuDoc* pDoc = GetDocument();
  ASSERT_VALID(pDoc);

  CMenu Menu;

  // расположение контекстного меню относительно экрана, нужно
  // преобразовать положение курсора мыши в экранные координаты
  ClientToScreen(&point);

  // создаем раскрывающееся меню
  Menu.CreatePopupMenu();
  for (int i = 0; i < NUM_COLOR; i++)
  {
    if (pDoc->m_aColor[i].m_btr == TRUE)
    {
      TCHAR strColor[64];
      sprintf(strColor, "Цвет %d", i);
      Menu.AppendMenu(MF_STRING  | MF_OWNERDRAW,
                      pDoc->m_aColor[i].m_nID, strColor);[1]
    }
  }

  // при помощи этого меню будем удалять элементы
  pDoc->bDelete = TRUE;

  // выводим контекстное меню на экран
  Menu.TrackPopupMenu(TPM_LEFTALIGN | TPM_RIGHTBUTTON,
                      point.x, point.y, this);
```

[1] Для режима самоотображения необходимо изменить эту функцию.

```
  // как всегда, не мешаем нормальной работе
  CView::OnRButtonDown(nFlags, point);
}

BOOL CDynMenuView::OnCmdMsg(UINT nID, int nCode, void* pExtra,
                            AFX_CMDHANDLERINFO* pHandlerInfo)
{
  CDynMenuDoc* pDoc = GetDocument();
  ASSERT_VALID(pDoc);

  if (pHandlerInfo == NULL)
  {
    // организуем фильтр для перехвата нужной команды
    for (int i = NUM_COLOR-1; i >= 0; i--)
    {
      if (nID == pDoc->m_aColor[i].m_nID)
      {
        if (nCode == CN_COMMAND)
        {
          // обрабатываем перехваченную команду

          // отдельная обработка для удаления элемента и его вставки
          if(pDoc->bDelete)
          {
            pDoc->bDelete = FALSE;
            pDoc->m_aColor[i].m_btr = FALSE;
          }
          else
            pDoc->DoSelectColor(i);

          // получаем доступ к фрейму
          CMainFrame* pwnd = (CMainFrame*)GetParentFrame();
          ASSERT_KINDOF(CMainFrame, pwnd);

          // обновление состава раскрывающегося меню
          pwnd->ChangeColorMenu();
          pDoc->UpdateAllViews(NULL);
        }
        else if (nCode == CN_UPDATE_COMMAND_UI)
        {
          // обновляем состояния элементов меню
          pDoc->DoUpdateSelectColor(i, (CCmdUI*)pExtra);
        }
        return TRUE;
      }
    }
  }

  // не забываем вернуть управление в стандартный маршрут команд
  return CView::OnCmdMsg(nID, nCode, pExtra, pHandlerInfo);
}
```

Создание собственных маркеров состояния

```
/*************************************************************
Файл MainFrm.h
Copyright (c) 1997 Мешков А., Тихомиров Ю., Шведов Д.
*************************************************************/
class CMainFrame : public CFrameWnd
{
...
// Attributes
public:
  CBitmap m_bmpCheck;      // для отмеченного состояния
  CBitmap m_bmpUnCheck;    // для неотмеченного состояния

  // ClassWizard generated virtual function overrides
  //{{AFX_VIRTUAL(CMainFrame)
  public:
    ...
    // изменять вид маркера будем здесь
    virtual void ActivateFrame(int nCmdShow = -1);
  //}}AFX_VIRTUAL
...
  DECLARE_MESSAGE_MAP()
};
/*************************************************************
Файл MainFrm.cpp
Copyright (c) 1997 Мешков А., Тихомиров Ю., Шведов Д.
*************************************************************/
void CMainFrame::ActivateFrame(int nCmdShow)
{
  // загружаем битовые массивы
  m_bmpCheck.LoadBitmap(IDB_MENU_DOWN);
  m_bmpUnCheck.LoadBitmap(IDB_MENU_UP);

  // получаем системные размеры маркера
  int cx = ::GetSystemMetrics(SM_CXMENUCHECK);
  int cy = ::GetSystemMetrics(SM_CYMENUCHECK);

  // приводим текущие размеры битовых массивов к системным
  m_bmpUnCheck.SetBitmapDimension(cx, cy);
  m_bmpCheck.SetBitmapDimension(cx, cy);

  CMenu *pViewMenu = NULL;
  CMenu *pTopMenu = AfxGetMainWnd()->GetMenu();
  int iPos;

  // ищем элемент меню "Добавить цвет" по идентификатору ID_ADD_COLOR
  for (iPos = pTopMenu->GetMenuItemCount()-1; iPos >= 0; iPos--)
```

```
  {
    CMenu* pMenu = pTopMenu->GetSubMenu(iPos);
    if (pMenu && pMenu->GetMenuItemID(0) == ID_VIEW_TOOLBAR)
    {
      pViewMenu = pMenu;
      break;
    }
  }
  ASSERT(pViewMenu != NULL);

  // устанавливаем новые маркеры для отмеченного и
  // неотмеченного состояний
  pViewMenu->SetMenuItemBitmaps(ID_VIEW_TOOLBAR, MF_BYCOMMAND,
                                &m_bmpUnCheck, &m_bmpCheck);
  pViewMenu->SetMenuItemBitmaps(ID_VIEW_STATUS_BAR, MF_BYCOMMAND,
                                &m_bmpUnCheck, &m_bmpCheck);

...

  // передаем управление фрейму
  CFrameWnd::ActivateFrame(nCmdShow);
}
```

Самоотображение элементов меню

```
/*************************************************************
Файл DynMenuView.h
Copyright (c) 1997 Мешков А., Тихомиров Ю., Шведов Д.
*************************************************************/

// для реализации режима самоотображения элемента меню необходимо
// создать свой класс на базе CMenu

class CColorMenu : public CMenu
{
public:
  CColorMenu(){}

// Implementation

  // переопределяем необходимые виртуальные функции
  virtual void MeasureItem(LPMEASUREITEMSTRUCT lpMIS);
  virtual void DrawItem(LPDRAWITEMSTRUCT lpDIS);
};

/*************************************************************
Файл DynMenuView.cpp
Copyright (c) 1997 Мешков А., Тихомиров Ю., Шведов Д.
*************************************************************/

// размеры для элемента меню в режиме самоотображения
#define COLOR_BOX_WIDTH     30
#define COLOR_BOX_HEIGHT    20
```

```
void CColorMenu::MeasureItem(LPMEASUREITEMSTRUCT lpMIS)
{
  // все элементы одного размера, но
  // при желании можно сделать и разные
  lpMIS->itemWidth = COLOR_BOX_WIDTH;
  lpMIS->itemHeight = COLOR_BOX_HEIGHT;
}

void CColorMenu::DrawItem(LPDRAWITEMSTRUCT lpDIS)
{
  // для любого графического вывода нужен контекст устройства
  CDC* pDC = CDC::FromHandle(lpDIS->hDC);

  COLORREF cr = (COLORREF)lpDIS->itemData; // цвет в элементе данных

  if (lpDIS->itemAction & ODA_DRAWENTIRE)
  {
    // рисуем элемент меню
    CBrush br(cr);
    pDC->FillRect(&lpDIS->rcItem, &br);
  }

  if ((lpDIS->itemState & ODS_SELECTED) &&
      (lpDIS->itemAction & (ODA_SELECT | ODA_DRAWENTIRE)))
  {
    // текущий элемент выделяем рамкой
    COLORREF crHilite = RGB(255-GetRValue(cr),
                            255-GetGValue(cr), 255-GetBValue(cr));
    CBrush br(crHilite);
    pDC->FrameRect(&lpDIS->rcItem, &br);
  }

  if (!(lpDIS->itemState & ODS_SELECTED) &&
      (lpDIS->itemAction & ODA_SELECT))
  {
    // невыделенный элемент - без рамки
    CBrush br(cr);
    pDC->FrameRect(&lpDIS->rcItem, &br);
  }
}
```

Полный текст рассмотренного приложения имеется на сопроводительной дискете.

Глава 30

Акселераторы, графические и другие ресурсы

Акселераторы

Для ускорения доступа к элементам меню при помощи клавиатуры, а также назначения тех или иных функций, не связанных с меню, комбинациям клавиш, в Windows используется так называемая таблица акселераторов (accelerator table). Она находится в ресурсах приложения и определяет соответствие между *акселератором* (комбинацией клавиш) и значением параметра WM_COMMAND, передающегося в функцию окна при их нажатии.

Для того чтобы некоторая комбинация клавиш стала работать как акселератор, она должна быть описана в таблице акселераторов и, кроме того, приложение должно загрузить таблицу акселераторов из ресурсов. Таблицу акселераторов можно создавать и изменять двумя путями — в файле ресурсов и с использованием некоторого визуального редактора ресурсов, типа AppStudio.[1]

Таблица акселераторов определяется в файле описания ресурсов приложения в следующем виде:

```
<ID> ACCELERATORS

BEGIN
  Key, ID, [Type], [Modifiers]
  ...
END
```

Идентификатор *ID* используется для ссылки на таблицу акселераторов, а *BEGIN* и *END* определяют операторные скобки, внутри которых располагаются строки описания акселераторов, имеющие следующий формат:

```
Key, ID, [Type], [Modifiers]
```

---

[1] В рассматриваемой версии Visual C++ такой редактор встроен в среду разработки.

Key — определяет клавишу, которая будет использована для создания акселератора. Можно использовать символ в коде ASCII, заключенный в двойные кавычки (например, "C"), комбинацию символа ASCII со знаком "^" (например, "^C", что соответствует комбинации клавиш <Ctrl>+<C>), код ASCII клавиши в виде целого значения или виртуальный код клавиши (в символьном или цифровом виде).

ID — задает идентификатор команды WM_COMMAND, которая направляется в функцию окна при использовании акселератора.

Type — принимает одно из двух значений: ASCII или VirtKey и определяет клавишу с использованием кода ASCII или с использованием кода виртуальной клавиши.

Modifiers — поле доступно, если в поле *Type* выбрано значение VirtKey; в этом случае можно указывать параметры Ctrl, Alt или Shift.

На рис. 65 изображено окно свойств для формирования таблицы акселераторов с использованием визуального средства построения ресурсов.

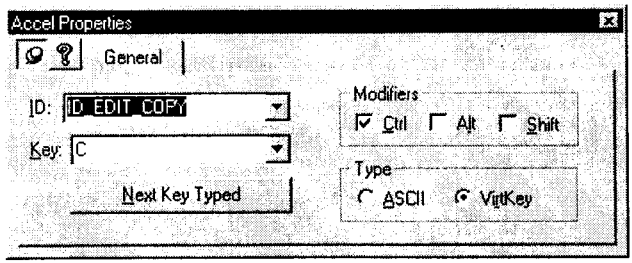


Рис. 65. Окно свойств, открытое в режиме настройки акселератора

После того как таблица определена, осталось совсем немного — приложение должно в определенном месте загрузить ее из файла ресурсов. Сделать это можно с помощью функции

BOOL CFrameWnd::**LoadAccelTable**(LPCTSTR lpszResourseName)

которая имеет единственный параметр *lpszResourceName*, определяющий имя таблицы акселераторов в файле ресурсов. Следует иметь в виду, что одновременно может быть загружена только одна таблица. При использовании числового идентификатора это имя можно получить при помощи макроса MAKEINTRESOURCE(lpszResourceName).

В случае, если создание окна производилось при помощи функции *LoadFrame*, таблица акселераторов загружается автоматически, одновременно с ресурсами меню и пиктограммы, которые должны иметь одинаковые идентификаторы. Ее уничтожение также производится библиотекой MFC.

Как видите, все очень просто. Единственное, что хотелось бы еще раз повторить: акселераторы представляют собой самостоятельную часть Windows

и позволяют не только дублировать команды меню, но и вызывать те команды, которые в нем отсутствуют.

Графические ресурсы

К графическим ресурсам Windows относятся *курсоры*, *пиктограммы*, *битовые массивы* и *панели инструментов*. Все они кодируются практически одинаково (различаясь только техникой использования) и являются частным случаем кодировки битовых массивов формата BMP. Первые три ресурса уже были достаточно подробно рассмотрены, поэтому здесь скажем несколько слов только о панелях инструментов. Они представляют собой набор обычных битовых массивов, каждому из которых соответствует определенный идентификатор команды.

На рис. 66 показан внешний вид окна свойств для настройки параметров кнопки панели инструментов, определяемых в полях:

ID определяет идентификатор команды, ассоциированной с данной кнопкой;

Width и Height — задают размеры кнопки (ширина и высота);

Prompt — содержит две строки, разделенные символом "\n", и изображаемые, соответственно, в строке состояния и во всплывающей подсказке.

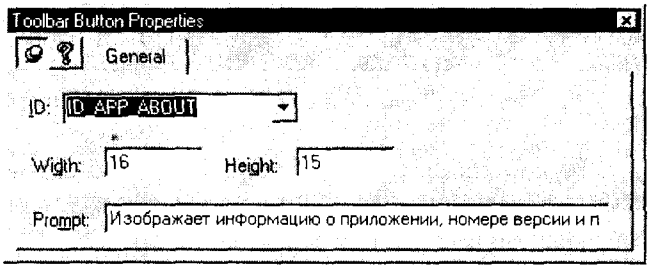


Рис. 66. Окно свойств, открытое для настройки кнопки панели инструментов

Подключение данного ресурса к фрейму осуществляется очень просто — в классе фрейма определяется переменная класса *CToolBar* для создания объекта:

```
class CMainFrame : public CFrameWnd
{
...
protected:  // control bar embedded members
  CToolBar    m_wndToolBar;

...
};
```

а в обработчике сообщения Windows WM_CREATE вызываются функции для создания окна Windows и загрузки панели инструментов из ресурсов:

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
  ...
  if (!m_wndToolBar.Create(this) ||              // создание окна
      !m_wndToolBar.LoadToolBar(IDR_MAINFRAME)) // загрузка ресурса
  {
    ...
    // обработка ошибки создания или загрузки панели инструментов
    return -1;
  }
  ...
}
```

Как видите, все точно так же, как и при "старом" способе создания панели инструментов. Отличие заключается в том, что теперь в тексте программы не надо привязывать кнопки к соответствующим идентификаторам — это делается в соответствующем ресурсе.

Строковые ресурсы

Это один из простейших ресурсов Windows. Как обычно, наиболее удобный способ построения заключается в использовании редактора ресурсов, где необходимо задать идентификатор строки и собственно ее текст. При этом в файл ресурсов добавляются строки вида:

```
STRINGTABLE DISCARDABLE
BEGIN
  ID_ADD_COLOR              "Добавить новый цвет\nЦвет"
  ID_DELETE_ALL             "Удалить все цвета\nУдалить цвета"
END
```

имеющие два параметра: идентификатор и собственно строка.

При сопоставлении строкового ресурса с некоторой командой она формируется автоматически при записи текста в поле Prompt соответствующего окна свойств.

Загрузка этого вида ресурса также очень проста, и заключается в использовании функции

BOOL CString::**LoadString**(UINT nID)

которая в качестве параметра использует идентификатор строкового ресурса.

При работе со строковыми ресурсами рекомендуем придерживаться следующих правил для экономии памяти:

- группировать строки по принципу совместного использования (например, строки, используемые в одном модуле приложения, имеют идентификаторы 100 — 107, а в другом модуле — 210 — 217);
- загружать строки в автоматические локальные переменные, чтобы используемое ими место автоматически освобождалось по окончании их использования;
- использовать одну и ту же переменную в качестве буфера хранения нескольких строк.

Ресурсы, определяемые программистом

Этот вид ресурсов используется для включения в исполняемый файл специальных данных, которые могут иметь произвольный формат. Для подключения таких ресурсов к приложению нужно описать их в файле ресурсов, например, так

```
UserResource USERDATA UserResource.dat
```

В процессе компоновки имя *UserResource* будет зарегистрировано как ресурс типа USERDATA в таблице ресурсов исполняемого файла. При этом для обозначения типа и имени ресурса могут использоваться числа (больше 255).

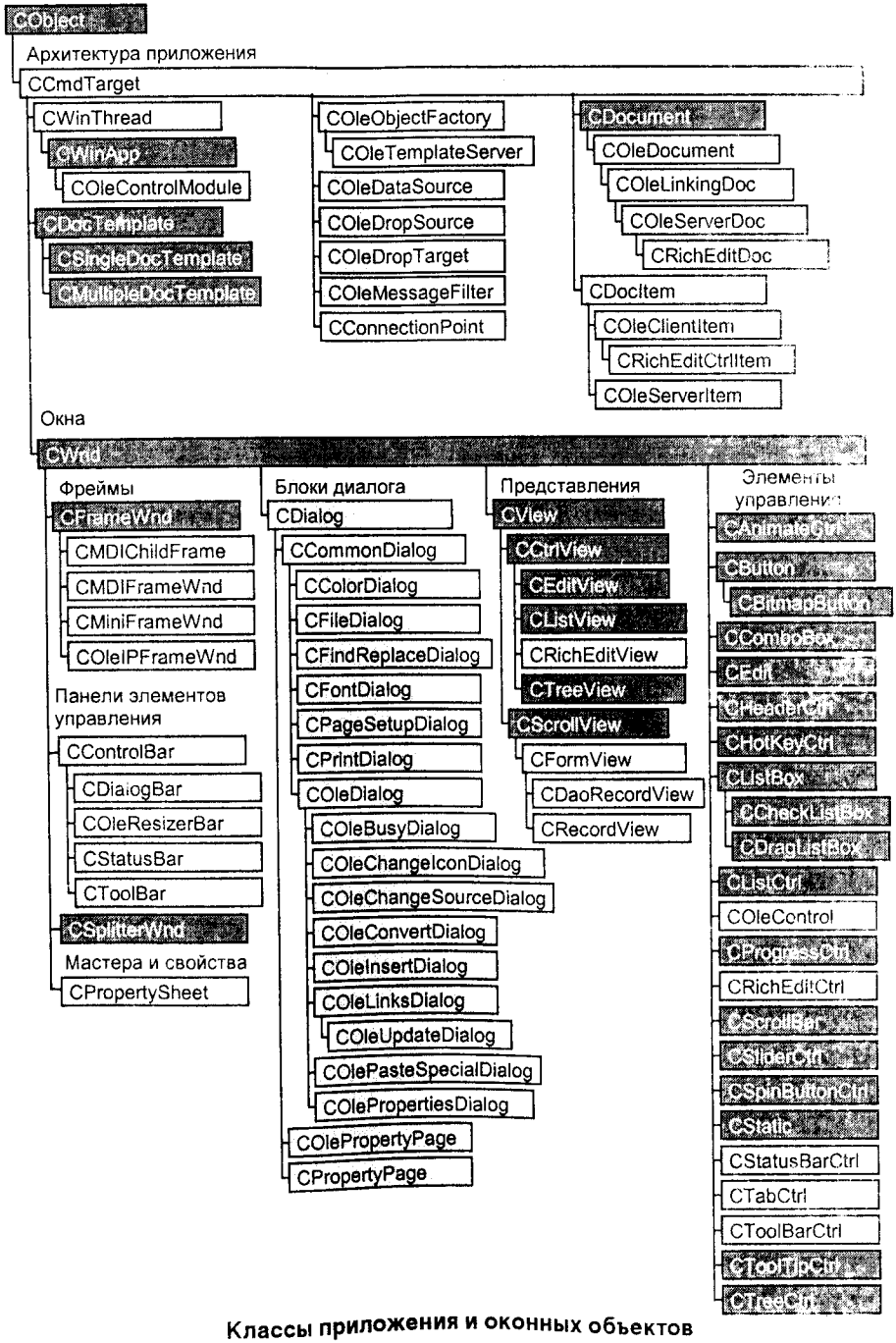
В библиотеке MFC отсутствует дополнительная поддержка этого типа ресурсов, поэтому для их использования следует обратиться к Win32 API, где определены следующие функции: *LoadResource*, *LockResource*, *FindResource*, *FindResourceEx*, *FreeResource*, *LoadLibrary* и *LoadModule*.

При работе с этими функциями есть два тонких момента. Первый заключается в том, что для загрузки ресурса необходимо вызвать функцию *LoadResource*, которая, однако, не загружает его в память, и работать с ним пока еще нельзя. Собственно загрузка осуществляется при закреплении блока данных в памяти функцией *LockResource*. А второй связан с тем, что в Windows NT ресурс освобождается операционной системой, а в Windows 95 это должен сделать сам программист, явно вызывая функцию *FreeResource*.

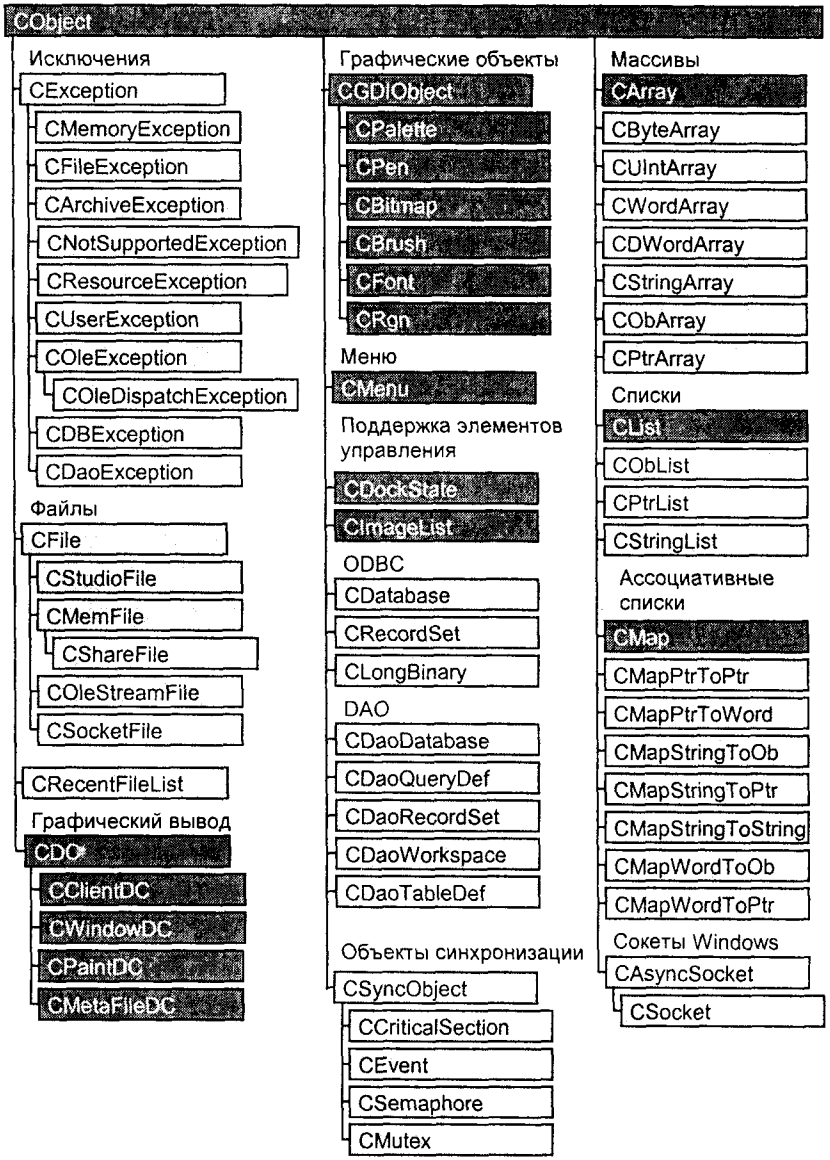
Приложения

1. Иерархия классов MFC 4.1
2. Функции-члены класса CDC
3. Значения параметра функции CDC::GetDeviceCaps

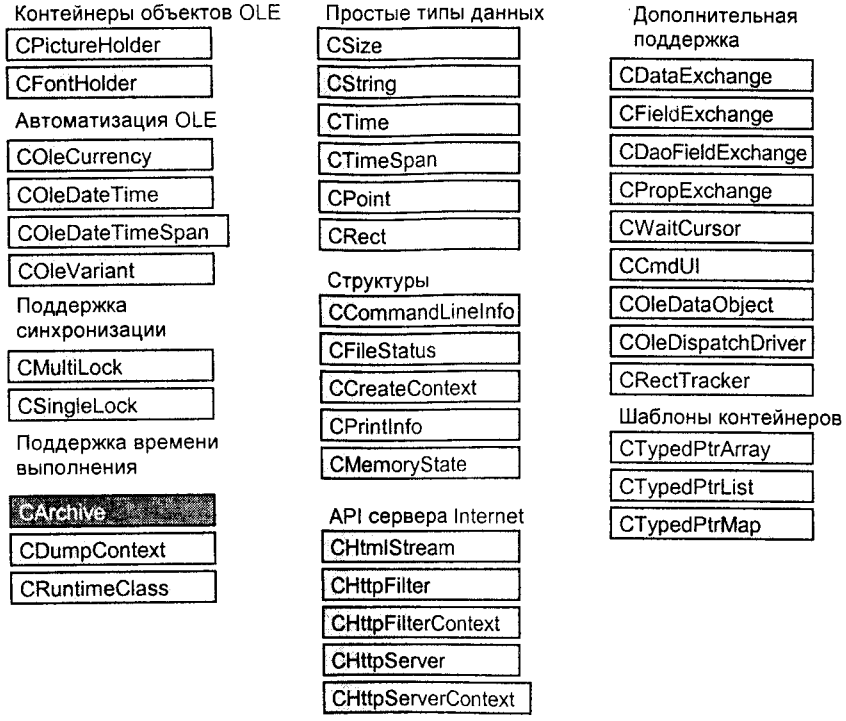
Приложение 1. Иерархия классов MFC 4.1



Классы приложения и оконных объектов



Классы неоконных объектов



Вспомогательные классы

Примечание. Черным цветом выделены классы, описанные в этом томе книги.

Приложение 2
Функции-члены класса CDC

Конструктор

CDC()	конструктор

Инициализация

virtual BOOL **CreateDC**(LPCTSTR lpszDriverName, LPCTSTR lpszDeviceName, LPCTSTR lpszOutput, const void* lpInitData)	создает контекст конкретного устройства
virtual BOOL **CreateIC**(LPCTSTR lpszDriverName, LPCTSTR lpszDeviceName, LPCTSTR lpszOutput, const void* lpInitData)	создает информационной контекст конкретного устройства для эффективного получения информации об устройстве без создания обычного контекста, которое требует существенных накладных расходов
virtual BOOL **CreateCompatibleDC**(CDC* pDC)	создает контекст устройства в памяти, совместимый с заданным
virtual BOOL **DeleteDC**()	удаляет контекст устройства, ассоциированный с объектом класса CDC
static CDC* PASCAL **FromHandle**(HDC hDC)	возвращает указатель на объект CDC, связанный с контекстом устройства дескриптором; при отсутствии такого объекта создает временный
static void PASCAL **DeleteTempMap**()	вызывается во время простоя объектом класса CWinApp для удаления всех временных объектов класса CDC и производных от него
BOOL **Attach**(HDC hDC)	связывает контекст устройства с объектом класса CDC
HDC **Detach**()	разрывает связь объекта класса CDC и контекста устройства
virtual void **SetAttribDC**(HDC hDC)	назначает для члена m_hAttribDC класса CDC дескриптор контекста устройства для получения параметров устройства

virtual void **SetOutputDC**(HDC hDC)	назначает для члена m_hDC класса CDC дескриптор контекста устройства
virtual void **ReleaseAttribDC**()	освобождает контекст устройства, определяемый дескриптором (m_hAttribDC)
virtual void **ReleaseOutputDC**()	освобождает контекст устройства, определяемый дескриптором (m_hDC)
CBitmap* **GetCurrentBitmap**()	возвращает указатель на установленный (выбранный) в данный момент объект класса CBitmap (битовый массив)
CBrush* **GetCurrentBrush**()	возвращает указатель на установленный (выбранный) в данный момент объект класса CBitmap (кисть)
CFont* **GetCurrentFont**()	возвращает указатель на установленный (выбранный) в данный момент объект класса CFont (шрифт)
CPalette* **GetCurrentPalette**()	возвращает указатель на установленный (выбранный) в данный момент объект класса CPalette (палитра)
CPen* **GetCurrentPen**()	возвращает указатель на установленный (выбранный) в данный момент объект класса CPen (карандаш)
CWnd* **GetWindow**()	возвращает указатель на объект класса CWnd — окно, ассоциированное с контекстом устройства

Работа с контекстом устройства

HDC **GetSafeHdc**()	возвращает значение члена m_hDC
virtual int **SaveDC**()	сохраняет текущие установки контекста устройства
virtual BOOL **RestoreDC**(int nSavedDC)	восстанавливает установки контекста устройства, ранее сохраненные функцией SaveDC
BOOL **ResetDC**(const DEVMODE* lpDevMode)	обновляет контекст устройства, определяемый членом m_hAttribDC
int **GetDeviceCaps**(int nIndex)	возвращает информацию о возможностях, предоставляемых устройством[1]
BOOL **IsPrinting**()	сообщает об использовании контекста устройства для печати

[1] Подробнее см. приложение 3.

Объекты рисования

CPoint **GetBrushOrg**()	возвращает точку привязки текущей кисти
CPoint **SetBrushOrg**(int x, int y) CPoint **SetBrushOrg**(POINT point)	устанавливает точку привязки текущей кисти
int **EnumObjects**(int nObjectType, int (CALLBACK EXPORT* lpfn) (LPVOID, LPARAM), LPARAM lpData)	позволяет получить информацию о доступных карандашах и кистях контекста устройства
CPen* **SelectObject**(CPen* pPen), CBrush* **SelectObject**(CBrush* pBrush), virtual CFont* **SelectObject**(CFont* pFont), CBitmap* **SelectObject**(CBitmap* pBitmap), int **SelectObject**(CRgn* pRgn)	устанавливают (выбирают) различные объекты рисования для дальнейшего их использования при выполнении операций рисования
virtual CGdiObject* **SelectStockObject**(int nIndex)	устанавливает (выбирает) предопределенные объекты рисования — карандаши, кисти и шрифты

Работа с цветом и цветовыми палитрами

COLORREF **GetNearestColor**(COLORREF crColor)	возвращает ближайший допустимый цвет, который возможно отобразить для контекста устройства, заданному параметром
CPalette* **SelectPalette**(CPalette* pPalette, BOOL bForceBackground)	устанавливает (выбирает) цветовую палитру
UINT **RealizePalette**()	сопоставляет элементы текущей палитры элементам системной палитры
void **UpdateColors**()	перерисовывает клиентскую область в соответствии с цветами, заданными в текущей палитре и сопоставленными с цветами системной палитры
static CBrush* PASCAL **GetHalftoneBrush**)	возвращает полутоновую кисть

Настройка параметров рисования

COLORREF **GetBkColor**()	возвращает текущий цвет фона
virtual COLORREF **SetBkColor**(COLORREF crColor)	устанавливает текущий цвет фона
int **GetBkMode**()	возвращает текущий режим отображения (прозрачный или нет)
int **SetBkMode**(int nBkMode)	устанавливает текущий режим отображения (прозрачный или нет)
int **GetPolyFillMode**()	возвращает текущий режим закрашивания многоугольников
int **SetPolyFillMode**(int nPolyFillMode)	устанавливает текущий режим закрашивания многоугольников
int **GetROP2**()	возвращает текущий режим рисования
int **SetBkMode**(int nBkMode)	устанавливает текущий режим рисования
int **GetStretchBltMode**()	возвращает текущий режим масштабирования битовых массивов
int **SetStretchBltMode**(int nStretchMode)	устанавливает текущий режим масштабирования битовых массивов
COLORREF **GetTextColor**()	возвращает текущий цвет, используемый для вывода текста
virtual COLORREF **SetTextColor**(COLORREF crColor)	устанавливает текущий цвет, используемый для вывода текста
BOOL **GetColorAdjustment**(LPCOLORADJUSTMENT lpColorAdjust)	возвращает значения настройки цветов устройства
BOOL **SetColorAdjustment**(const COLORADJUSTMENT* lpColorAdjust)	устанавливает значения настройки цветов устройства

Настройка режимов отображения

int **GetMapMode**()	возвращает текущий режим отображения
virtual int **SetMapMode**(int nMapMode)	устанавливает текущий режим отображения
CPoint **GetViewportOrg**()	возвращает оконные координаты точки начала координат физической области вывода

virtual CPoint **SetViewportOrg**(int x, int y), virtual CPoint **SetViewportOrg**(POINT point)	устанавливают оконные координаты точки начала координат физической области вывода
CPoint **OffsetWindowOrg**(int nWidth, int nHeight)	изменяет начало координат физической области вывода относительно его текущего положения
CSize **GetViewportExt**()	возвращает размеры физической области вывода
virtual CSize **SetViewportExt**(int cx, int cy), virtual CSize **SetViewportExt**(SIZE size)	устанавливают размеры физической области вывода
virtual CSize **ScaleViewportExt**(int xNum, int xDenom, int yNum, int yDenom)	изменяет размеры физической области вывода относительно их текущих значений
CPoint **GetWindowOrg**()	возвращает оконные координаты точки начала координат логической области вывода
CPoint **SetWindowOrg**(int x, int y), CPoint **SetWindowOrg**(POINT point)	устанавливают оконные координаты точки начала координат логической области вывода
CPoint **OffsetWindowOrg**(int nWidth, int nHeight)	изменяет начало координат логической области вывода относительно его текущего положения
CSize **GetWindowExt**()	возвращает размеры логической области вывода
virtual CSize **SetWindowExt**(int cx, int cy), virtual CSize **SetWindowExt**(SIZE size)	устанавливают размеры логической области вывода
virtual CSize **ScaleWindowExt**(int xNum, int xDenom, int yNum, int yDenom)	изменяет размеры логической области вывода относительно их текущих значений

Преобразование координат

void **DPtoHIMETRIC**(LPSIZE lpSize)	преобразует единицы устройства в логические единицы режима HIMETRIC
void **DPtoLP**(LPPOINT lpPoints, int nCount = 1), void **DPtoLP**(LPRECT lpRect), void **DPtoLP**(LPSIZE lpSize),	преобразуют единицы устройства в логические единицы текущего режима отображения
void **HIMETRICtoDP**(LPSIZE lpSize)	преобразует логические единицы режима HIMETRIC в единицы устройства
void **HIMETRICtoLP**(LPSIZE lpSize)	преобразует логические единицы режима HIMETRIC в логические единицы текущего режима отображения
void **LPtoDP**(LPPOINT lpPoints, int nCount = 1), void **LPtoDP**(LPRECT lpRect), void **LPtoDP**(LPSIZE lpSize)	преобразуют логические единицы текущего режима отображения в единицы устройства
void **LPtoHIMETRIC**(LPSIZE lpSize)	преобразует логические единицы текущего режима отображения в логические единицы режима HIMETRIC

Работа с регионами

BOOL **FillRgn**(CRgn* pRgn, CBrush* pBrush)	заполняет заданный регион при помощи заданной кисти
BOOL **FrameRgn**(CRgn* pRgn, CBrush* pBrush, int nWidth, int nHeight)	рисует рамку вокруг заданного региона при помощи заданной кисти
BOOL **InvertRgn**(CRgn* pRgn)	инвертирует содержимое региона
BOOL **PaintRgn**(CRgn* pRgn)	заполняет заданный регион при помощи текущей кисти

Работа с областью отсечения

UINT **SetBoundsRect**(LPCRECT lpRectBounds, UINT flags)	изменяет область отсечения
UINT **GetBoundsRect**(LPRECT lpRectBounds, UINT flags)	возвращает минимальную прямоугольную область отсечения
virtual int **GetClipBox**(LPRECT lpRect)	возвращает размеры области отсечения
virtual int **SelectClipRgn**(CRgn* pRgn) int **SelectClipRgn**(CRgn* pRgn, int nMode)	изменяет текущую область отсечения с учетом заданного региона и режима комбинирования
virtual int **ExcludeClipRect**(int x1, int y1, int x2, int y2) virtual int **ExcludeClipRect**(LPCRECT lpRect)	изменяет текущую область отсечения, исключая из нее заданный прямоугольник
int **ExcludeUpdateRgn**(CWnd* pWnd)	предотвращает рисование в области отсечения
virtual int **IntersectClipRect**(int x1, int y1, int x2, int y2), virtual int **IntersectClipRect**(LPCRECT lpRect)	создают новую область отсечения, являющуюся пересечением текущей области и заданного прямоугольника
virtual int **OffsetClipRgn**(int x, int y), virtual int **OffsetClipRgn**(SIZE size)	перемещают текущую область отсечения
virtual BOOL **PtVisible**(int x, int y), virtual BOOL **PtVisible**(POINT point)	определяют нахождение заданной точки в области отсечения
virtual BOOL **RectVisible**(LPCRECT lpRect)	определяет, пересекается ли заданный прямоугольник с текущей областью отсечения

Рисование линий

CPoint **GetCurrentPosition**()	возвращает текущее положение для карандаша в логических единицах
CPoint **MoveTo**(int x, int y), CPoint **MoveTo**(POINT point)	перемещают текущее положение карандаша
CPoint **LineTo**(int x, int y), CPoint **LineTo**(POINT point)	рисуют линию от текущего положения до заданной точки, не включая последнюю
BOOL **Arc**(int x1, int y1, int x2, int y2, int x3, int y3, int x4, int y4), BOOL **Arc**(LPCRECT lpRect, POINT ptStart, POINT ptEnd)	рисуют дугу
BOOL **ArcTo**(int x1, int y1, int x2, int y2, int x3, int y3, int x4, int y4), BOOL **ArcTo**(LPCRECT lpRect, POINT ptStart, POINT ptEnd)	рисуют дугу с одновременным изменением текущего положения карандаша
BOOL **AngleArc**(int x, int y, int nRadius, float fStartAngle, float fSweepAngle)	рисует сегмент с одновременным изменением текущего положения карандаша
int **GetArcDirection**()	возвращает текущее направление рисования дуг
int **SetArcDirection**(int nArcDirection)	устанавливает текущее направление рисования дуг
BOOL **PolyDraw**(const POINT* lpPoints, const BYTE* lpTypes, int nCount)	рисует ломаные и кривые Безье с одновременным изменением текущего положения карандаша
BOOL **Polyline**(LPPOINT lpPoints, int nCount)	рисует ломаную линию

BOOL **PolyPolyline**(const POINT* lpPoints, const DWORD* lpPolyPoints, int nCount)	рисует совокупность ломаных линий
BOOL **PolylineTo**(const POINT* lpPoints, int nCount)	рисует ломаную линию с одновременным изменением текущего положения карандаша
BOOL **PolyBezier**(const POINT* lpPoints, int nCount)	рисует кривую Безье
BOOL **PolyBezierTo**(const POINT* lpPoints, int nCount)	рисует кривую Безье с одновременным изменением текущего положения карандаша

Простейшее рисование изображений

void **FillRect**(LPCRECT lpRect, CBrush* pBrush)	заполняет заданный прямоугольник при помощи заданной кисти
void **FrameRect**(LPCRECT lpRect, CBrush* pBrush)	рисует рамку вокруг заданного прямоугольника при помощи заданной кисти
void **InvertRect**(LPCRECT lpRect)	инвертирует содержимое прямоугольника
BOOL **DrawIcon**(int x, int y, HICON hIcon), BOOL **DrawIcon**(POINT point, HICON hIcon)	рисуют пиктограмму
void **DrawDragRect**(LPCRECT lpRect, SIZE size, LPCRECT lpRectLast, SIZE sizeLast, CBrush* pBrush = NULL, CBrush* pBrushLast = NULL)	рисует и удаляет прямоугольник, имитируя операцию перемещения (drag)
void **FillSolidRect**(LPCRECT lpRect, COLORREF clr) void **FillSolidRect**(int x, int y, int cx, int cy, COLORREF clr)	заполняет прямоугольник заданным цветом

void **Draw3dRect**(LPCRECT lpRect, COLORREF clrTopLeft, COLORREF clrBottomRight), void **Draw3dRect**(int x, int y, int cx, int cy, COLORREF clrTopLeft, COLORREF clrBottomRight)	рисуют трехмерный прямоугольник
BOOL **DrawEdge**(LPRECT lpRect, UINT nEdge, UINT nFlags)	рисует выпуклую (вогнутую) прямоугольную рамку
BOOL **DrawFrameControl**(LPRECT lpRect, UINT nType, UINT nState)	рисует элементы управления (кнопки, пункты меню и т. д.)
BOOL **DrawState**(...)[1]	рисует изображения и пиктограммы с возможностью отображения ряда состояний

Рисование эллипсов и многоугольников

BOOL **Chord**(int x1, int y1, int x2, int y2, int x3, int y3, int x4, int y4), BOOL **Chord**(LPCRECT lpRect, POINT ptStart, POINT ptEnd)	рисуют сегмент
void **DrawFocusRect**(LPCRECT lpRect)	рисует прямоугольник, аналогичный рамке визуализации фокуса
BOOL **Ellipse**(int x1, int y1, int x2, int y2), BOOL **Ellipse**(LPCRECT lpRect)	рисуют эллипс
BOOL **Pie**(int x1, int y1, int x2, int y2, int x3, int y3, int x4, int y4), BOOL **Pie**(LPCRECT lpRect, POINT ptStart, POINT ptEnd)	рисуют сектор

[1] Эта функция имеет 8 различных версий, которые подробно описаны в главе 17.

BOOL **Polygon**(LPPOINT lpPoints, int nCount)	рисует многоугольник
BOOL **PolyPolygon**(LPPOINT lpPoints, LPINT lpPolyCounts, int nCount)	рисует совокупность многоугольников
BOOL **Rectangle**(int x1, int y1, int x2, int y2), BOOL **Rectangle**(LPCRECT lpRect)	рисуют прямоугольник
BOOL **RoundRect**(int x1, int y1, int x2, int y2, int x3, int y3), BOOL **RoundRect**(LPCRECT lpRect, POINT point)	рисуют прямоугольник с закругленными углами

Рисование изображений

BOOL **PatBlt**(int x, int y, int nWidth, int nHeight, DWORD dwRop)	создает изображение в результате комбинирования текущей кисти и содержимого области вывода
BOOL **BitBlt**(int x, int y, int nWidth, int nHeight, CDC* pSrcDC, int xSrc, int ySrc, DWORD dwRop)	копирует битовый массив из заданого контекста устройства
BOOL **StretchBlt**(int x, int y, int nWidth, int nHeight, CDC* pSrcDC, int xSrc, int ySrc, int nSrcWidth, int nSrcHeight, DWORD dwRop)	копирует битовый массив из заданного контекста устройства с возможностью масштабирования

COLORREF **GetPixel**(int x, int y), COLORREF **GetPixel**(POINT point)	возвращают цвет заданной точки
COLORREF **SetPixel**(int x, int y, COLORREF crColor), COLORREF **SetPixel**(POINT point, COLORREF crColor)	рисуют заданную точку цветом, наиболее близким к заданному
BOOL **SetPixelIV**(int x, int y, COLORREF crColor), BOOL **SetPixelIV**(POINT point, COLORREF crColor)	рисуют заданную точку цветом, наиболее близким к заданному быстрее, чем функция SetPixel, т. к. не возвращает предыдущее значение цвета
BOOL **FloodFill**(int x, int y, COLORREF crColor)	заполняет область при помощи заданной кисти
BOOL **ExtFloodFill**(int x, int y, COLORREF crColor, UINT nFillType)	заполняет область при помощи заданной кисти, обеспечивая больше возможностей, чем функция FloodFill
BOOL **MaskBlt**(int x, int y, int nWidth, int nHeight, CDC* pSrcDC, int xSrc, int ySrc, CBitmap & maskBitmap, int xMask, int yMask, DWORD dwRop)	комбинирует цвета точек двух битовых массивов, используя маску и учитывая заданную растровую операцию
BOOL **PlgBlt**(POINT lpPoint, CDC* pSrcDC, int xSrc, int ySrc, int nWidth, int nHeight, CBitmap & maskBitmap, int xMask, int yMask)	обеспечивает перемещение битового массива в заданный по точкам параллелограмм, обеспечивая, например, вращение

Отображение текстовых строк

virtual BOOL **TextOut**(int x, int y, LPCTSTR lpszString, int nCount), BOOL **TextOut**(int x, int y, const CString & str)	отображают строку текста
virtual BOOL **ExtTextOut**(int x, int y, UINT nOptions, LPCRECT lpRect, LPCTSTR lpszString, UINT nCount, LPINT lpDxWidths), BOOL **ExtTextOut**(int x, int y, UINT nOptions, LPCRECT lpRect, const CString & str, LPINT lpDxWidths)	отображают строку текста в рамках заданной прямоугольной области
virtual CSize **TabbedTextOut**(int x, int y, LPCTSTR lpszString, int nCount, int nTabPositions, LPINT lpnTabStopPositions, int nTabOrigin), CSize **TabbedTextOut**(int x, int y, const CString & str, int nTabPositions, LPINT lpnTabStopPositions, int nTabOrigin)	отображают строку текста в заданной позиции, раскрывая символы табуляции с учетом позиций, заданных в массиве
virtual int **DrawText**(LPCTSTR lpszString, int nCount, LPRECT lpRect, UINT nFormat), int **DrawText**(const CString & str, LPRECT lpRect, UINT nFormat)	отображают строку текста в заданной прямоугольной области с ее форматированием

CSize **GetTextExtent**(LPCTSTR lpszString, int nCount), CSize **GetTextExtent**(const CString & str)	вычисляют ширину и высоту строки текста, которые она будет иметь при отображении на устройстве
CSize **GetOutputTextExtent**(LPCTSTR lpszString, int nCount), CSize **GetOutputTextExtent**(const CString & str)	вычисляют ширину и высоту строки текста, которые она будет иметь при отображении на устройстве вывода
CSize **GetTabbedTextExtent**(LPCTSTR lpszString, int nCount, int nTabPositions, LPINT lpnTabStopPositions), CSize **GetTabbedTextExtent**(const CString & str, int nTabPositions, LPINT lpnTabStopPositions)	вычисляют ширину и высоту строки текста (с учетом знаков табуляции), которые она будет иметь при отображении на устройстве
CSize **GetOutputTabbedTextExtent**(LPCTSTR lpszString, int nCount, int nTabPositions, LPINT lpnTabStopPositions), CSize **GetOutputTabbedTextExtent**(const CString & str, int nTabPositions, LPINT lpnTabStopPositions)	вычисляют ширину и высоту строки текста (с учетом знаков табуляции), которые она будет иметь при отображении на устройстве вывода
virtual BOOL **GrayString**(CBrush* pBrush, BOOL (CALLBACK EXPORT* lpfnOutput) (HDC, LPARAM, int), LPARAM lpData, int nCount, int x, int y, int nWidth, int nHeight)	отображает строку текста серым цветом
UINT **GetTextAlign**()	возвращает флаги выравнивания текста
UINT **SetTextAlign**(UINT nFlags)	устанавливает флаги выравнивания текста
int **GetTextFace**(int nCount, LPTSTR lpszFacename), int **GetTextFace**(CString& rString)	копируют имя текущего шрифта в буфер

BOOL **GetTextMetrics**(LPTEXTMETRIC lpMetrics)	заполняет структуру *TEXTMETRICS* информацией о текущем шрифте контекста устройства
BOOL **GetOutputTextMetrics**(LPTEXTMETRIC lpMetrics)	заполняет структуру *TEXTMETRICS* информацией о текущем шрифте контекста устройства вывода
int **SetTextJustification**(int nBreakExtra, int nBreakCount)	раздвигает символы строки текста для ее выравнивания по ширине
int **GetTextCharacterExtra**()	возвращает текущие установки для межсимвольных интервалов
int **SetTextCharacterExtra**(int nCharExtra)	устанавливает текущие установки для межсимвольных интервалов

Работа со шрифтами

DWORD **GetFontData**(DWORD dwTable, DWORD dwOffset, LPVOID lpData, DWORD cbData)	возвращает параметры масштабируемого шрифта
int **GetKerningPairs**(int nPairs, LPKERNINGPAIR lpkrnpair)	возвращает пары кернинга для текущего шрифта устройства
UINT **GetOutlineTextMetrics**(UINT cbData, LPOUTLINETEXTMETRIC lpotm)	возвращает параметры шрифтов TrueType
DWORD **GetGlyphOutline**(UINT nChar, UINT nFormat, LPGLYPHMETRICS lpgm, DWORD cbBuffer, LPVOID lpBuffer, const MAT2 FAR* lpmat2)	возвращает кривые отрисовки или битовый массив символа текущего шрифта
BOOL **GetCharABCWidths**(UINT nFirstChar, UINT nLastChar, LPABC lpabc), BOOL **GetCharABCWidths**(UINT nFirstChar, UINT nLastChar, LPABCFLOAT lpABCF)	возвращают ширину заданной последовательности символов текущего шрифта устройства

BOOL **GetCharWidth**(UINT nFirstChar, UINT nLastChar, LPINT lpBuffer), BOOL **GetCharWidth**(UINT nFirstChar, UINT nLastChar, float* lpFloatBuffer)	возвращают ширину каждого символа из заданной последовательности символов текущего шрифта устройства
BOOL **GetOutputCharWidth**(UINT nFirstChar, UINT nLastChar, LPINT lpBuffer)	возвращает размер каждого символа из заданной последовательности символов текущего шрифта устройства вывода
DWORD **SetMapperFlags**(DWORD dwFlag)	задает алгоритм отображения логических шрифтов на устройстве
CSize **GetAspectRatioFilter**()	возвращает текущие установки фильтра масштабирования

Управление устройствами

BOOL **QueryAbort**()	вызывает функцию AbortProc приложения и запрашивает возможность приостановки печати
virtual int **Escape**(int nEscape, int nCount, LPCSTR lpszInData, LPVOID lpOutData), int **ExtEscape**(int nEscape, int nInputSize, LPCSTR lpszInputData, int nOutputSize, LPSTR lpszOutputData)	позволяют приложению воспользоваться специфическими возможностями устройства, которые не поддерживаются функциями GDI
int **DrawEscape**(int nEscape, int nInputSize, LPCSTR lpszInputData)	позволяет приложению воспользоваться специфическими возможностями видеосистемы, которые не поддерживаются функциями GDI
int **StartDoc**(LPDOCINFO lpDocInfo)	информирует драйвер устройства о поступлении нового задания на печать
int **StartPage**()	информирует драйвер устройства о начале печати новой страницы документа
int **EndPage**()	информирует драйвер устройства о завершении печати страницы документа

int **SetAbortProc**(BOOL (CALLBACK EXPORT* lpfn) (HDC, int))	задает функцию прерывания печати
int **AbortDoc**()	прекращает выполнение текущего задания на печать и удаляет то, что было получено драйвером устройства со времени последнего вызова функции StartDoc
int **EndDoc**()	завершает задание на печать

Прокрутка

BOOL **ScrollDC**(int dx, int dy, LPCRECT lpRectScroll, LPCRECT lpRectClip, CRgn* pRgnUpdate, LPRECT lpRectUpdate)	осуществляет прокрутку содержимого прямоугольной области по горизонтали или по вертикали

Работа с метафайлами

BOOL **PlayMetaFile**(HMETAFILE hMF),	"проигрывают" содержимое метафайла
BOOL **PlayMetaFile**(HENHMETAFILE hEnhMetaFile, LPCRECT lpBounds)	
BOOL **AddMetaFileComment**(UINT nDataSize, const BYTE* pCommentData)	добавляет комментарий из заданного буфера к расширенному метафайлу

Работа с контурами

BOOL **AbortPath**()	прекращает "запись" контура с одновременным удалением всей ранее сохраненной о нем информацией
BOOL **BeginPath**()	указывает GDI, что последующие вызовы функций рисования должны не отображаться, а использоваться для создания контура
BOOL **CloseFigure**()	замыкает контур
BOOL **EndPath**()	прекращает накапливание информации о контуре
BOOL **FillPath**()	замыкает текущий контур и заполняет его внутреннюю область с помощью текущей кисти и режима заполнения многоугольников

BOOL **FlattenPath**()	трансформирует все кривые контуры в наборы прямых отрезков
float **GetMiterLimit**()	возвращает расстояние, используемое для соединения концевых точек линий
int **GetPath**(LPPOINT lpPoints, LPBYTE lpTypes, int nCount)	возвращает информацию о точках и элементах, определяющих контур
BOOL **SelectClipPath**(int nMode)	трансформирует контур в регион, комбинирует полученный таким образом регион с текущей областью отсечения в определенном параметром режиме
BOOL **SetMiterLimit**(float fMiterLimit)	устанавливает расстояние, используемое для соединения концевых точек линий
BOOL **StrokeAndFillPath**()	замыкает текущий контур, заполняет его внутреннюю область с помощью текущей кисти, отображая сам контур с помощью текущего карандаша
BOOL **StrokePath**()	отображает контур с помощью текущего карандаша
BOOL **WidenPath**()	назначает область, определяемую контуром областью, требующей перерисовки, при выполнении отображения линий контура

Приложение 3
Значения параметра функции CDC::GetDeviceCaps

Значение параметра	Возвращаемая информация об устройстве
DRIVERVERSION	версия драйвера, например, 0x0204 означает версию 2.04
TECHNOLOGY	тип устройства; устройства могут быть следующих типов: DT_PLOTTER — векторный плоттер DT_RASDISPLAY — растровый дисплей DT_RASPRINTER — растровый принтер DT_RASCAMERA — растровая камера DT_CHARSTREAM — поток символов DT_METAFILE — метафайл DT_DISPFILE — образ экрана в файле
HORZSIZE	ширина в миллиметрах
VERTSIZE	высота в миллиметрах
HORZRES	ширина в пикселях
VERTRES	высота в пикселях
LOGPIXELSX	число пикселей на дюйм по горизонтали
LOGPIXELSY	число пикселей на дюйм по вертикали
BITSPIXEL	количество бит информации для кодировки цвета одного пикселя
PLANES	количество цветовых слоев
NUMBRUSHES	число кистей, определенных для устройства
NUMPENS	число карандашей, определенных для устройства
NUMFONTS	число шрифтов, определенных для устройства
NUMCOLORS	число цветов, определенных в таблице устройства
ASPECTX	относительная ширина точки устройства, используемая для рисования линий
ASPECTY	относительная высота точки устройства, используемая для рисования линий
ASPECTXY	относительная ширина по диагонали точки устройства, используемая для рисования линий

(продолжение)

Значение параметра	Возвращаемая информация об устройстве
PDEVICESIZE	размер внутренней структуры *PDEVICE*
CLIPCAPS	возможности отсечения, поддерживаемые устройством; устройства могут поддерживать одну из следующих возможностей: CP_NONE — отсечение не поддерживается CP_RECTANGLE — отсечения задаются прямоугольными областями CP_REGION — отсечения задаются регионами
SIZEPALETTE	количество элементов системной палитры устройства; значение верно, если устройство поддерживает палитры (бит RC_PALETTE установлен в ответ на запрос RASTERCAPS)
NUMRESERVED	количество свободных (зарезервированных) элементов системной палитры устройства; значение верно, если устройство поддерживает палитры (бит RC_PALETTE установлен в ответ на запрос RASTERCAPS)
COLORRES	фактическое количество бит, определяющих цвет пикселя для устройства; значение верно, если устройство поддерживает палитры (бит RC_PALETTE установлен в ответ на запрос RASTERCAPS)
RASTERCAPS	растровые возможности устройства; устройство может предоставлять одну или несколько из следующих возможностей: RC_BANDING — поддержка прокрутки RC_BIGFONT — поддержка шрифтов размером более 64K RC_BITBLT — поддержка копирования битовых массивов RC_BITMAP64 — поддержка битовых массивов размером больше 64K RC_DEVBITS — использование системных битовых массивов RC_DI_BITMAP — поддержка функций *SetDIBits* и *GetDIBits* RC_DIBTODEV — поддержка функции *SetDIBitsToDevice* RC_FLOODFILL — поддержка сплошной заливки контуров RC_GDI20_OUTPUT — поддержка графических возможностей Windows 2.0 RC_GDI20_STATE — включение блока состояния в структуре контекста устройства RC_NONE — нет поддержки растровых операций RC_OP_DX_OUTPUT — поддержка непрозрачного режима вывода

(продолжение)

Значение параметра	Возвращаемая информация об устройстве	
	RC_PALETTE	требуется задание палитры цветов
	RC_SAVEBITMAP	поддержка локального сохранения битовых массивов
	RC_SCALING	поддержка масштабирования
	RC_STRETCHBLT	поддержка функции *StretchBlt*
	RC_STRETCHDIB	поддержка функции *StretchDIBits*
CURVECAPS	поддержка рисования кривых; устройство может предоставлять одну или несколько из следующих возможностей:	
	CC_NONE	поддержка отсутствует
	CC_CIRCLES	поддержка рисования окружностей
	CC_PIE	поддержка рисования секторов
	CC_CHORD	поддержка рисования сегментов
	CC_ELLIPSES	поддержка рисования эллипсов
	CC_WIDE	поддержка рисования линий различной толщины
	CC_STYLED	поддержка рисования линий различных стилей
	CC_WIDESTYLED	поддержка рисования широких линий различных стилей
	CC_INTERIORS	поддержка заливки внутренних областей замкнутых кривых
	CC_ROUNDRECT	поддержка рисования прямоугольников с округлыми углами
LINECAPS	поддержка рисования прямых линий; устройство может предоставлять одну или несколько из следующих возможностей:	
	LC_NONE	поддержка отсутствует
	LC_POLYLINE	поддержка рисования ломаных
	LC_MARKER	поддержка рисования маркеров линий
	LC_POLYMARKER	поддержка рисования маркеров ломаных
	LC_WIDE	поддержка рисования широких линий
	LC_STYLED	поддержка рисования линий различных стилей
	LC_WIDESTYLED	поддержка рисования широких линий различных стилей
	LC_INTERIORS	поддержка заливки внутренних областей замкнутых ломаных

(продолжение)

Значение параметра	Возвращаемая информация об устройстве	
POLYGONALCAPS	поддержка рисования замкнутых фигур, состоящих из прямых линий; устройство может предоставлять одну или несколько из следующих возможностей:	
	PC_NONE	поддержка отсутствует
	PC_POLYGON	поддержка различных видов заливки
	PC_RECTANGLE	поддержка рисования прямоугольников
	PC_WINDPOLYGON	поддержка заливки сложных многоугольников
	PC_WIDE	поддержка рисования широких линий
	PC_STYLED	поддержка рисования линий различных стилей
	PC_WIDESTYLED	поддержка рисования широких линий различных стилей
	PC_INTERIORS	поддержка заливки внутренних областей замкнутых фигур
TEXTCAPS	поддержка вывода текста; устройство может предоставлять одну или несколько из следующих возможностей:	
	TC_OP_CHARACTER	поддержка возможности отображения символов шрифтов в любом месте устройства с точностью в один пиксель
	TC_OP_STROKE	поддержка настройки точности места отображения системных шрифтов
	TC_CP_STROKE	поддержка возможности вывода символьной информации с задаваемых границ
	TC_CR_90	поддержка возможности поворота текста ровно на 90^0 за один раз
	TC_CR_ANY	поддержка возможности поворота текста на любой угол
	TC_SF_X_YINDEP	поддержка независимого масштабирования по горизонтали и по вертикали
	TC_SA_DOUBLE	поддержка двойного увеличения символов при масштабировании
	TC_SA_INTEGER	поддержка увеличения в целое число раз величины символов при масштабировании
	TC_SA_CONTIN	поддержка масштабирования с любым коэффициентом

(продолжение)

Значение параметра	Возвращаемая информация об устройстве	
	TC_EA_DOUBLE	поддержка рисования полужирных символов; если этот бит не установлен для принтеров, GDI пытается выводить полужирные символы, пропечатывая их дважды
	TC_IA_ABLE	поддержка вывода текста курсивом
	TC_UA_ABLE	поддержка вывода подчеркнутого текста; если этот бит не установлен, GDI подчеркивает текст, выводя линию под ним
	TC_SO_ABLE	поддержка вывода зачеркнутого текста; если этот бит не установлен, GDI зачеркивает текст, выводя линию по тексту
	TC_RA_ABLE	поддержка возможности перечисления растровых шрифтов и шрифтов TrueType устройства при помощи функций *EnumFonts* и *EnumFontFamilies*
	TC_VA_ABLE	поддержка возможности перечисления векторных шрифтов устройства при помощи функций *EnumFonts* и *EnumFontFamilies*; проверка этого бита не требуется для драйверов дисплея и растровых принтеров, т. к. GDI преобразует векторные шрифты в растры перед посылкой их драйверу
	TC_RESERVED	зарезервировано, должно быть равно 0

Предметный указатель

A

ACN_START 186
ACN_STOP 186
ACS_AUTOPLAY 185
ACS_CENTER 185
ACS_TRANSPARENT 185
AfxRegisterWndClass 38
ALTERNATE 22
AnsiToOem 117, 143
ANSI_CHARSET 45, 61
ANSI_FIXED_FONT 20
ANSI_VAR_FONT 20
ARABIC_CHARSET 45, 61

B

BALTIC_CHARSET 45, 61
BLACK_BRUSH 20
BLACK_PEN 20
BLACKNESS 36
BLACKONWHITE 23
BS_3STATE 110
BS_AUTO3STATE 110
BS_AUTOCHECKBOX 109
BS_AUTORADIOBUTTON 109
BS_BITMAP 110
BS_BOTTOM 110
BS_CENTER 110
BS_CHECKBOX 109
BS_DEFPUSHBUTTON 109
BS_FLAT 110
BS_GROUPBOX 109
BS_ICON 110
BS_LEFT 110
BS_LEFTTEXT 110
BS_MULTILINE 110
BS_NOTIFY 110
BS_OWNERDRAW 110, 158
BS_PUSHBUTTON 109
BS_PUSHLIKE 110
BS_RADIOBUTTON 109
BS_RIGHT 110
BS_RIGHTBUTTON 110
BS_TEXT 110
BS_TOP 110
BS_VCENTER 110

C

CAnimateCtrl
Close 185
Create 185
Open 185
Play 186
Seek 186
Stop 186
CArchive 318
Abort 318
bNoFlushOnDelete 318
Close 318
Flush 319
IsLoading 320
IsStoring 320
load 318
m_pDocument 317
Read 319
ReadString 319
store 318
Write 319
WriteString 320
CBitmap
CreateBitmap 66, 71
CreateBitmapInderect 66
CreateCompatibleBitmap 20, 66, 70
CreateDiscardableBitmap 66
GetBitmap 69
GetBitmapBits 67
GetBitmapDimension 51, 67, 68
LoadBitmap 67
LoadMappedBitmap 67
LoadOEMBitmap 67
HBITMAP 51
SetBitmapBits 67, 71
SetBitmapDimension 51, 67, 68
CBitmapButton
AutoLoad 159, 160
LoadBitmaps 160
SizeToContent 160
CBN_CLOSEUP 142
CBN_EDITUPDATE 141
CBN_SELENDCANCEL 142
CBN_SELENDOK 142
CBrush 58
CreateBrushIndirect 58
CreateDIBPatternBrush 58
CreateHatchBrush 58, 93
CreatePatternBrush 58
CreateSolidBrush 58, 93
CreateSysColorBrush 58
HBRUSH 51
CBS_AUTOHSCROLL 143
CBS_DISABLENOSCROLL 144
CBS_DROPDOWN 141, 143, 146
CBS_DROPDOWNLIST 141, 143, 146

CBS_HASSTRINGS 143
CBS_NOINTEGRALHEIGHT 144
CBS_OEMCONVERT 143
CBS_OWNERDRAWFIXED 143
CBS_OWNERDRAWVARIABLE 143
CBS_SIMPLE 141, 143
CBS_SORT 144
CButton
Create 111, 160
DrawItem 114
GetButtonStyle 113
GetCheck 112
GetState 112
SetBitmap 114
SetButtonStyle 113
SetCheck 113
SetCursor 114
SetIcon 114
SetState 112
CCheckListBox
Create 161
DrawItem 163
Enable 162
GetCheck 162
GetCheckStyle 162
IsEnabled 162
MeasureItem 163
OnGetCheckPosition 162
SetCheck 162
SetCheckStyle 162
CComboBox
AddString 135, 148
Clear 144
Copy 144
Create 142
Cut 144
DeleteString 148
Dir 148
DrawItem 148
FindString 148
FindStringExact 148
GetCount 145
GetCurSel 145
GetDroppedControlRect 146
GetDroppedState 146
GetDroppedWidth 146
GetEditSel 144
GetExtendedUI 147
GetHorizontalExtent 145
GetItemData 145
GetItemDataPtr 145
GetItemHeight 145
GetLBText 142, 147
GetLBTextLen 147
GetLocale 146
GetText 143
GetTopIndex 145
InitStorage 146
InsertString 135, 148
LimitText 144
MeasureItem 148
Paste 144
ResetContent 148
SelectString 148
SetCurSel 145
SetDroppedWidth 146
SetEditSel 146
SetExtendedUI 147
SetHorizontalExtent 145
SetItemData 145
SetItemDataPtr 145
SetItemHeight 145
SetLocale 146
SetTopIndex 145
ShowDropDown 146
CControlBar
SetBarStyle 218
CCS_BOTTOM 233
CCS_NOPARENTALIGN 233
CCS_NORESIZE 233
CCS_TOP 233
CCtrlView
CCtrlView 328
m_dwDefaultStyle 327
m_strClass 327
CDC
AddMetaFileComment 46
AngleArc 30, 33
Arc 30, 33, 86
ArcTo 30, 33
BeginPath 33, 89
BitBlt 37
Chord 33, 53
CloseFigure 33
CreateCompatibleDC 16
CreateDC 16
DeleteTempMap 18
DPtoHIMETRIC 30
DPtoLP 30
Draw3dRect 31
DrawEdge 31
DrawEscape 46
DrawFocusRect 33
DrawFrameControl 31
DrawIcon 31
DrawState 31, 32, 72
DrawText 41
Ellipse 33, 53, 86
EndDoc 46
EndPage 46
EndPath 33, 89
EnumObjects 19
Escape 46
ExcludeClipRect 30

ExcludeUpdateRgn 30
ExtFloodFill 39, 87
ExtTextOut 24, 33, 40
FillPath 33, 89
FillRect 31, 92
FillRgn 30
FillSolidRect 31
FloodFill 39
FrameRect 31
FrameRgn 30
FromHandle 17
GetBrushOrg 19
GetCharABCWidth 45
GetCharWidth 45
GetClipBox 30
GetCurrentBitmap 18
GetCurrentBrush 18
GetCurrentFont 18
GetCurrentPalette 18
GetCurrentPen 18
GetDeviceCaps 17, 18
GetFontData 45
GetGlyphOutline 45
GetHalftoneBrush 21
GetKerningPairs 45
GetOutlineTextMetrics 45
GetOutputCharWidth 45
GetOutputTextExtent 40
GetOutputTextMetrics 43
GetPixel 36
GetSafeHdc 18
GetTextAllign 40
GetTextExtent 40
GetTextFace 40
GetTextMetrics 43, 90
HIMETRICtoDP 30
HIMETRICtoLP 30
IntersectClipRect 30
InvertRect 31
InvertRgn 30
LineTo 30, 33, 53, 87
LPtoDP 30, 87
LPToHIMETRIC 30
m_hAttribDC 15
m_hDC 15
MoveTo 33, 87
PaintRgn 30, 88
PatBlt 36
Pie 33
PlayMetaFile 46
PlgBlt 36
PolyBezier 30, 33, 89
PolyBezierTo 30, 33
PolyDraw 30, 33
Polygon 33, 91
Polyline 30, 33
PolylineTo 30, 33
PolyPolygon 33
PolyPolyline 30, 33
PtVisible 30
Rectangle 33, 53
RestoreDC 18
RoundRect 33, 53, 86
SaveDC 18
ScrollDC 39
SelectClipPath 35
SelectClipRgn 20, 30, 35, 87
SelectObject 19, 86
SelectPalette 21
SelectStockObject 20, 50
SetAttribDC 15
SetBkColor 24, 89
SetBkMode 21, 90
SetBrushOrg 19
SetMapMode 25, 85
SetMiterLimit 55
SetOutputDC 15
SetPixel 36
SetPixelV 36
SetPolyFillMode 21
SetROP2 22
SetStretchBltMode 23
SetTextAlign 40, 41, 43, 89
SetTextColor 24
SetTextJustification 40
SetViewportExt 28, 85
SetViewportOrg 26, 85
SetWindowExt 28, 85
SetWindowOrg 26
StartDoc 46
StartPage 46
StokePath 33
StretchBlt 38
StretchBltMode 38
StrokeAndFillPath 33
TabbedTextOut 40
TextOut 24, 33, 40, 90
UpdateColors 21

CDialog
DoModal 181
OnInitDialog 181
OnOK 181

CDocManager
m_templateList 296

CDocTemplate 287
AddDocument 287
CloseAllDocuments 289
CreateNewDocument 290
CreateNewFrame 290
docName 288
fileNewName 288
filterExt 289
filterNewName 289
GeNextDoc 292

GetDocString 288
GetFirstDocPosition 292
InitialUpdateFrame 291
LoadTemplate 287
MatchDocType 290
OpenDocumentFile 292
regFileTypeId 289
regFileTypeName 289
RemoveDocument 287
SaveAllModified 289
SetDefaultTitle 292
windowTitle 288
CDocument 309
AddView 310
CanCloseFrame 312
DeleteContents 312
GetDocTemplate 309
GetFile 314
GetFirstViewPosition 310
GetNextView 310
GetPathName 310
GetTitle 310
IsModified 310
OnChangedViewList 313
OnCloseDocument 313
OnNewDocument 313
OnOpenDocument 313
OnSaveDocument 313
PreCloseFrame 314
ReleaseFile 314
RemoveView 310
ReportSaveLoadException 314
SaveModified 314
SetModifiedFlag 310
SetPathName 310
SetTitle 310
UpdateAllViews 311
CDragListBox
BeginDrag 171
CancelDrag 171
Dragging 172
DrawInsert 171
DrawSingle 171
Dropped 172
ItemFromPt 171
m_nLast 171
CEdit
CanUndo 121, 122
CharFromPos 124
Clear 121
Copy 121
Create 116
Cut 121
EmptyUndoBuffer 122
GetFirstVisibleLine 119
GetHandle 121
GetLimitText 124
GetLine 120
GetLineCount 120
GetMargins 123
GetModify 121, 122
GetPasswordChar 123
GetRect 123
GetSel 118
LimitText 123
LineFromChar 120
LineIndex 120
LineLength 120
LineScroll 119
Paste 121
PosFromChar 124
ReplaceSel 119
SetHandle 121, 122
SetLimitText 123, 124
SetMargins 124
SetModify 122
SetPasswordChar 123
SetReadOnly 123
SetRect 123
SetRectNP 123
SetSel 118
Undo 121, 122
CEditView 329
dwStyleDefault 329
FindText 330
GetBufferLength 330
GetPrinterFont 329
GetSelectedText 330
LockBuffer 330
OnFindText 330
OnReplaceAll 331
OnReplaceSel 331
OnTextNotFound 331
PrintInsideRect 329
SerializeRaw 330
SetPrinterFont 329
SetTabStops 329
UnlockBuffer 330
CF_TEXT 121, 144
CFont
CreateFont 59, 64, 93
CreateFontIndirect 59
CreatePointFont 59, 65
CreatePointFontIndirect 59, 65
HFONT 51
CFrameWnd
LoadAccelTable 409
OnEndPrintPreview 369
OnSetPreviewMode 369
CGdiObject
Attach 50, 67
CreateStockObject 50, 58
DeleteObject 48, 88
FromHandle 48

GetObject 51, 69
GetSafeHandle 47
m_hObject 47
HGDIOBJ 51
CHeaderCtrl
Create 233
DeleteItem 235
DrawItem 238
GetItem 235
GetItemCount 235
InsertItem 235
SetItem 235
CHINESEBIG5_CHARSET 45, 61
CHotKeyCtrl
Create 189
GetHotKey 190, 191, 193
SetHotKey 190
SetRules 191
CImageList
Add 225
Attach 224, 226
BeginDrag 227
Create 224, 225
Detach 224, 226
DragEnter 227
DragLeave 228
DragMove 227
DragShowNolock 228
Draw 226
EndDrag 228
ExtractIcon 225
GetImageInfo 231
GetSafeHandle 224
m_hImageList 224
Read 226
Remove 225
Replace 225
SetBkColor 226
SetDragCursorImage 228
SetOverlayImage 227
Write 226
CLIP_CHARACTER_PRECIS 63
CLIP_DEFAULT_PRECIS 63
CLIP_EMBEDDED 63
CLIP_LH_ANGLES 63
CLIP_MASK 63
CLIP_STROKE_PRECIS 63
CLIP_TT_ALWAYS 63
CListBox
AddString 129, 131, 135
CompareItem 130, 136
Create 125
DeleteString 131
Dir 130, 131
DrawItem 134, 137
FindString 131
FindStringExact 132
GetCount 132
GetCurSel 132
GetHorizontalExtent 127
GetItemData 133
GetItemDataPtr 133
GetItemHeight 128
GetItemRect 127
GetLocale 129
GetSel 132
GetSelItems 132, 133
GetText 133
GetTextLen 133
GetTopIndex 133
InitStorage 131
InsertString 130, 131, 135
ItemFromPoint 134
MeasureItem 135, 137
ResetContent 131
SelectString 131
SetColumnWidth 125, 127, 128
SetCurSel 125, 132
SetHorizontalExtent 127
SetItemData 133
SetItemDataPtr 133
SetItemHeight 128
SetItemRange 133
SetLocale 129
SetSel 132
SetTabStops 128
SetTopIndex 133
CListCtrl
Arrange 243
Create 240
CreateDragImage 258
DeleteAllItems 246
DeleteColumn 255
DeleteItem 248
DrawItem 258
EditLabel 256
EnsureVisible 243
FindItem 251
GetBkColor 242
GetCallbackMask 246
GetColumn 255
GetColumnWidth 255
GetCountPerPage 243
GetEditControl 256
GetImageList 244
GetItem 248
GetItemCount 246
GetItemData 249
GetItemPosition 250
GetItemRect 250
GetItemState 249
GetItemText 249
GetNextItem 250
GetOrigin 243

GetSelectedCount 246
GetStringWidth 254
GetTextBkColor 242
GetTextColor 242
GetTopIndex 242
GetViewRect 242
HitTest 252
InsertColumn 254
InsertItem 247
RedrawItems 243
Scroll 243
SetBkColor 242
SetCallbackMask 246
SetColumn 255
SetColumnWidth 255
SetImageList 244
SetItem 248
SetItemCount 246
SetItemData 249
SetItemPosition 250
SetItemState 249
SetItemText 249
SetTextBkColor 242
SetTextColor 242
SortItems 253
Update 243
CListView
CListView 334
GetListCtrl 334
CLR_NONE 226, 242
CMenu 378
AppendMenu 135, 380
CheckMenuItem 383
CheckMenuRadioItem 383
CreateMenu 378
CreatePopupMenu 378
DeleteMenu 382
DestroyMenu 380
DrawItem 385
EnableMenuItem 382
GetMenuContextHelpId 386
GetMenuItemCount 383
GetMenuItemID 384
GetMenuState 384
GetMenuString 384
GetSubMenu 384
InsertMenu 135, 381
LoadMenu 378
LoadMenuIndirect 379
m_hMenu 378
MeasureItem 385
ModifyMenu 135, 382
RemoveMenu 382
SetMenuContextHelpId 386
SetMenuItemBitmaps 383
TrackPopupMenu 384
CMultiDocTemplate
CMultiDocTemplate 293
COLORONCOLOR 23
COMPAREITEMSTRUCT 137
CPalette
HPALETTE 51
CPen 52
CreatePen 58, 93
CreatePenIndirect 58
HPEN 51
CProgressCtrl
Create 200
OffsetPos 201
SetPos 200
SetRange 200
SetStep 201
StepIt 200
CreateDIBitmap 72
CRgn
CombineRgn 76, 88
CopyRgn 77
CreateEllipticRgn 75
CreateEllipticRgnIndirect 75, 88
CreateFromPath 77
CreatePolygonRgn 75
CreatePolyPolygonRgn 75
CreateRectRgn 74
CreateRectRgnIndirect 74, 87
CreateRoundRectRgn 75
EqualRgn 77
HRGN 51
CRichEditCtrl 180
CScrollBar
Create 151
EnableScrollBar 154
GetScrollInfo 155
GetScrollLimit 156
GetScrollPos 153
GetScrollRange 153
SetScrollBar 150
SetScrollInfo 155
SetScrollPos 154
SetScrollRange 153
ShowScrollBar 154
CScrollView 335
GetDeviceScrollPosition 337
GetDeviceScrollSizes 336
GetScrollPosition 337
GetTotalSize 336
ResizeParentToFit 339
ScrollToPosition 337
SetScaleToFitSize 337
SetScrollSizes 335
CSingleDocTemplate 291
CSliderCtrl
ClearSel 208
ClearTics 209
Create 206
GetChannelRect 208

GetLineSize 208
GetNumTics 208
GetPageSize 208
GetPos 209
GetRange 208
GetRangeMax 208
GetRangeMin 208
GetSelection 208
GetThumbRect 208
GetTic 208
GetTicArray 208
SetLineSize 208
SetPageSize 208
SetPos 209
SetRange 208
SetRangeMax 208
SetRangeMin 208
SetSelection 208
SetTicFreq 209
VerifyPos 209
CSpinButtonCtrl
Create 212
GetBase 215
GetBuddy 213
GetPos 213
GetRange 214
SetAccel 214
SetBase 215
SetBuddy 213
SetPos 213
SetRange 214
CSplitterWnd 342
ActivateNext 348
CanActivateNext 348
Create 342
CreateScrollBarCtrl 348
CreateStatic 343
CreateView 344
DeleteColumn 347
DeleteRow 347
DeleteView 347
DoKeyboardSplit 348
DoScroll 348
DoScrollBy 348
GetActivePane 347
GetColumnCount 345
GetColumnInfo 345
GetPane 345
GetRowCount 345
GetRowInfo 345
GetScrollStyle 346
IdFromEowCol 346
IsChildPane 346
OnDrawSplitter 346
OnInvertTracker 347
RecalcLayout 346
SetActivePane 347
SetColumnInfo 345
SetRowInfo 345
SetScrollStyle 346
SplitRow 347
CSrollBar
SetScrollRange 156
CStatic 100
Create 101
GetBitmap 103
GetCursor 105
GetEnhMetaFile 105
GetIcon 104
SetBitmap 103
SetCursor 105
SetEnhMetaFile 105
SetIcon 105
CString
LoadString 411
CToolBar
Create 218
CToolBarCtrl
GetToolTips 218
SetToolTips 218
CToolTip
Create 220
CToolTipCtrl
Activate 221
AddTool 220
DelTool 221
GetText 222
GetToolInfo 222
HitTest 222
RelayEvent 222
SetDelayTime 222
SetToolInfo 222
UpdateTipText 222
CTreeCtrl
Create 267
CreateDragImage 280
DeleteAllItems 269
DeleteItem 272
EditLabel 280
EnsureVisible 276
Expand 275
GetChildItem 274
GetCount 269
GetDropHilightItem 277
GetEditControl 280
GetFirstVisibleItem 277
GetImageList 268
GetIndent 268
GetItem 273
GetItemData 274
GetItemImage 273
GetItemRect 274
GetItemState 273
GetItemText 273

GetNextItem 276
GetNextSiblingItem 276
GetNextVisibleItem 277
GetParentItem 276
GetPrevSiblingItem 276
GetPrevVisibleItem 277
GetRootItem 277
GetSelectedItem 277
GetVisibleCount 269
HitTest 277
InsertItem 269
ItemHasChildren 274
Select 275
SelectDropTarget 275
SelectItem 275
SetImageList 268
SetIndent 268
SetItem 272
SetItemData 274
SetItemImage 273
SetItemState 273
SetItemText 274
SortChildren 278
SortChildrenCB 278
CTreeView 334
GetTreeCtrl 334
CView 323
DoPreparePrinting 357
DoPrintPreview 368
GetDocument 323
OnActivateFrame 325
OnActivateView 325
OnBeginPrinting 358
OnDraw 8, 326
OnEndPrinting 359
OnFilePrint 360
OnInitialUpdate 324
OnPrepareDC 8, 359
OnPreparePrinting 355
OnPrint 360
OnScroll 326
OnScrollBy 326
OnUpdate 324
CWinApp
AddDocTemplate 287, 294
CloseAllDocuments 296
CreatePrinterDC 354
GetFirstDocTemplatePosition 295
GetNextDocTemplate 295
GetPrinterDeviceDefaults 353
m_pDocManager 296
OnFileNew 296
OnFileOpen 296
OnFilePrintSetup 351
OpenDocumentFile 295
SaveAllModified 296
CWnd
Default 150
DestroyWindow 181
DrawMenuBar 383
EnableToolTips 216, 218
GetCurrentMessage 149
GetStyle 123
GetWindowText 142
OnCommand 182
OnHScroll 149
OnInitMenu 389
OnInitMenuPopup 389
OnMenuSelect 390
OnNotify 182
OnVScroll 150
SetScrollPos 150
SetWindowText 119, 122, 123

D

DEFAULT_CHARSET 45, 61
DEFAULT_PALETTE 21
DEFAULT_PITCH 64
DEFAULT_QUALITY 63
DEVICE_DEFAULT_FONT 20
DKGRAY_BRUSH 20, 21
DL_COPYCURSOR 172
DL_MOVECURSOR 172
DL_STOPCURSOR 172
DRAFT_QUALITY 63
DRAWITEMSTRUCT 134
DSS_DEFAULT 33
DSS_DISABLED 33
DSS_MONO 33
DSS_NORMAL 32
DSS_UNION 32
DST_BITMAP 32
DST_COMPLEX 32
DST_ICON 32
DST_PREFIXTEXT 32
DST_TEXT 32
DST_INVERT 36
DT_BOTTOM 41
DT_CALCRECT 41
DT_CENTER 41
DT_EXPANDTABS 42
DT_EXTERNALLEADING 42
DT_LEFT 42
DT_NOCLIP 42
DT_NOPREFIX 42
DT_RIGHT 42
DT_SINGLELINE 42
DT_TABSTOP 42
DT_TOP 42
DT_VCENTER 42
DT_WORDBREAK 42, 43

E

EASTEUROPE_CHARSET 45, 61
ES_AUTOHSCROLL 116
ES_AUTOVSCROLL 116
ES_CENTER 116
ES_LEFT 116
ES_LOWERCASE 116
ES_MULTILINE 117
ES_NOHIDESEL 117
ES_OEMCONVERT 117
ES_PASSWORD 117
ES_READONLY 117
ES_RIGHT 117
ES_UPPERCASE 117
ES_WANTRETURN 117
ESB_DISABLE_BOTH 154
ESB_DISABLE_LTUP 154
ESB_DISABLE_RTDN 154
ESB_ENABLE_BOTH 154
ETO_CLIPPED 40
ETO_OPAQUE 41

F

FF_DECORATIVE 45, 64
FF_DONTCARE 45, 64
FF_MODERN 45, 64
FF_ROMAN 45, 64
FF_SCRIPT 45, 64
FF_SWISS 45, 64
FIXED_PITCH 64
FLOODFILLBORDER 39
FLOODFILLSURFACE 39
FW_BLACK 61
FW_BOLD 61
FW_DEMIBOLD 61
FW_DONTCARE 61
FW_EXTRABOLD 61
FW_EXTRALIGHT 61
FW_HEAVY 61
FW_LIGHT 61
FW_MEDIUM 61
FW_NORMAL 61
FW_REGULAR 61
FW_SEMIBOLD 61
FW_THIN 61
FW_ULTRABOLD 61
FW_ULTRALIGHT 61

G

GB2312_CHARSET 45, 61
GetDialogBaseUnits 126
GetWindowLong 242
GlobalAddAtom 193
GM_ADVANCED 61
GRAY_BRUSH 20, 21
GREEK_CHARSET 45, 61

H

HALFTONE 23
HANGEUL_CHARSET 45, 61
HD_ITEM 234
HD_NOTIFY 236
HDF_BITMAP 234
HDF_CENTER 234
HDF_LEFT 234
HDF_OWNERDRAW 234
HDF_RIGHT 234
HDF_STRING 234
HDI_BITMAP 234
HDI_FORMAT 234
HDI_HEIGHT 234
HDI_LPARAM 234
HDI_TEXT 235
HDI_WIDTH 235
HDN_BEGINTRACK 236
HDN_DIVIDERDBLCLICK 235
HDN_ENDTRACK 236
HDN_ITEMCHANGED 235
HDN_ITEMCHANGING 235
HDN_ITEMCLICK 235
HDN_ITEMDBLCLICK 235
HDN_TRACK 236
HDS_BUTTONS 233
HDS_HIDDEN 233
HDS_HORZ 233
HDS_VERT 233
HEBREW_CHARSET 45, 61
HKCOMB_A 191
HKCOMB_C 191
HKCOMB_CA 191
HKCOMB_NONE 191
HKCOMB_S 191
HKCOMB_SA 191
HKCOMB_SC 191
HKCOMB_SCA 191
HOLLOW_BRUSH 20
HOTKEYF_ALT 190
HOTKEYF_CONTROL 190
HOTKEYF_EXT 190
HOTKEYF_SHIFT 190
HS_BDIAGONAL 58
HS_CROSS 58
HS_DIAGCROSS 58
HS_FDIAGONAL 59
HS_HORIZONTAL 59
HS_VERTICAL 59

I

ILD_BLEND25 227
ILD_BLEND50 226
ILD_NORMAL 226
ILD_OVERLAYMASK 227
ILD_TRANSPARENT 226
IMAGEINFO 231
INDEXTOOVERLAYMASK 227

J

JOHAB_CHARSET 45, 61

L

LBS_DISABLENOSCROLL 126
LBS_EXTENDEDSEL 125
LBS_HASSTRINGS 125, 130, 132, 136, 170, 172
LBS_MULTICOLUMN 125
LBS_MULTIPLESEL 126, 131, 170
LBS_NOINTEGRALHEIGHT 126, 137
LBS_NOREDRAW 126
LBS_NOTIFY 124, 126
LBS_OWNERDRAWFIXED 126, 134, 170, 172
LBS_OWNERDRAWVARIABLE 126, 128, 134, 137, 170, 172
LBS_SORT 126, 129, 130, 170
LBS_STANDARD 126
LBS_USETABSTOPS 126
LBS_WANTKEYBOARDINPUT 125, 126
LocalAlloc 121
LocalFree 121
LOGFONT 60
LTGRAY_BRUSH 20, 21
LV_COLUMN 254
LV_HITTESTINFO 252
LV_ITEM 247
LVA_ALIGNLEFT 243
LVA_ALIGNTOP 243
LVA_DEFAULT 243
LVA_SNAPTOGRID 243
LVA_SORTASCENDING 244
LVA_SORTDESCENDING 244
LVCF_FMT 255
LVCF_SUBITEM 255
LVCF_TEXT 255
LVCF_WIDTH 255
LVFI_NEARESTXY 252
LVFI_PARAM 251
LVFI_PARTIAL 251
LVFI_STRING 252
LVFI_WRAP 252
LVHT_ABOVE 253
LVHT_BELOW 253
LVHT_NOWHERE 253
LVHT_ONITEM 253
LVHT_ONITEMICON 253
LVHT_ONITEMLABEL 253
LVHT_ONITEMSTATEICON 253
LVHT_TOLEFT 253
LVHT_TORIGHT 253
LVIF_IMAGE 247
LVIF_PARAM 247
LVIF_STATE 247
LVIF_TEXT 247
LVIR_BOUNDS 250
LVIR_ICON 250
LVIR_LABEL 250
LVIS_CUT 246, 248
LVIS_DROPHILITED 246, 248
LVIS_FOCUSED 246, 248
LVIS_OVERLAYMASK 246
LVIS_SELECTED 246, 248
LVIS_STATEIMAGEMASK 246
LVN_BEGINDRAG 256, 257
LVN_BEGINLABELEDIT 256
LVN_BEGINRDRAG 256, 257
LVN_COLUMNCLICK 256
LVN_DELETEALLITEMS 256
LVN_DELETEITEM 256
LVN_ENDLABELEDIT 241, 256
LVN_GETDISPINFO 256
LVN_INSERTITEM 257
LVN_ITEMCHANGED 257
LVN_ITEMCHANGING 257
LVN_KEYDOWN 257
LVN_SETDISPINFO 257
LVNI_ABOVE 251
LVNI_ALL 251
LVNI_BELOW 251
LVNI_DROPHILITED 251
LVNI_FOCUSED 251
LVNI_HIDDEN 251
LVNI_MARKED 251
LVNI_SELECTED 251
LVNI_TOLEFT 251
LVNI_TORIGHT 251
LVS_ALIGNLEFT 241
LVS_ALIGNTOP 241
LVS_AUTOARRANGE 241
LVS_EDITLABELS 241
LVS_ICON 241
LVS_LIST 241
LVS_NOCOLUMNHEADER 241
LVS_NOLABELWRAP 241
LVS_NOSCROLL 241
LVS_NOSORTHEADER 241
LVS_OWNERDRAWFIXED 241
LVS_REPORT 241
LVS_SHAREIMAGELISTS 241

LVS_SINGLESEL 241
LVS_SMALLICON 241
LVS_SORTASCENDING 241
LVS_SORTDESCENDING 241

M

MAC_CHARSET 45, 61
MCIWnd 183
MEASUREITEMSTRUCT 136
MENUITEMTEMPLATE 379
MENUITEMTEMPLATEHEADER 379
MERGECOPY 37
MERGEPAINT 37
MF_BITMAP 390
MF_BYCOMMAND 381
MF_BYPOSITION 382
MF_CHECKED 380, 390
MF_DISABLED 380, 390
MF_ENABLED 380
MF_GRAYED 380, 390
MF_MENUBARBREAK 380
MF_MENUBREAK 380
MF_MOUSESELECT 390
MF_OWNERDRAW 380, 381, 390
MF_POPUP 380, 390
MF_SEPARATOR 380, 381, 390
MF_STRING 380, 381
MF_SYSMENU 390
MF_UNCHECKED 380
MM_ANISOTROPIC 25, 26
MM_HIENGLISH 25
MM_HIMETRIC 25
MM_ISOTROPIC 25, 26
MM_LOENGLISH 25
MM_LOMETRIC 25
MM_TEXT 24, 25, 60
MM_TWIPS 25
MOD_ALT 193
MOD_CONTROL 193
MOD_SHIFT 193

N

NM_LISTVIEW 257
NM_TREEVIEW 280
NM_UPDOWNW 211
NOTSRCCOPY 37
NOTSRCERASE 37
NULL_BRUSH 20
NULL_PEN 20

O

OBM_BTNCORNERS 67
OBM_BTSIZE 67
OBM_CHECK 67
OBM_CHECKBOXES 67
OBM_CLOSE 67
OBM_COMBO 67
OBM_DNARROW 67
OBM_DNARROWD 67
OBM_DNARROWI 67
OBM_LFARROW 67
OBM_LFARROWD 67
OBM_LFARROWI 67
OBM_MNARROW 67
OBM_OLD_CLOSE 67
OBM_OLD_DNARROW 67
OBM_OLD_LFARROW 67
OBM_OLD_REDUCE 67
OBM_OLD_RESTORE 67
OBM_OLD_RGARROW 67
OBM_OLD_UPARROW 67
OBM_OLD_ZOOM 67
OBM_REDUCE 67
OBM_REDUCED 67
OBM_RESTORE 67
OBM_RESTORED 67
OBM_RGARROW 67
OBM_RGARROWD 67
OBM_RGARROWI 67
OBM_SIZE 67
OBM_UPARROW 67
OBM_UPARROWD 67
OBM_UPARROWI 67
OBM_ZOOM 67
OBM_ZOOMD 67
ODA_DRAWENTIRE 135
ODA_FOCUS 135
ODA_SELECT 135
ODS_DISABLED 135
ODS_FOCUS 135
ODS_SELECTED 135
ODT_BUTTON 135
ODT_COMBOBOX 135
ODT_LISTBOX 135
ODT_MENU 135
OEM_CHARSET 45, 61
OEM_FIXED_FONT 20
ON_BN_CLICKED 111
ON_BN_DOUBLECLICKED 111
ON_CBN_CLOSEUP 141
ON_CBN_DBLCLK 141
ON_CBN_DROPDOWN 141
ON_CBN_EDITCHANGE 141
ON_CBN_EDITUPDATE 142
ON_CBN_ERRSPACE 142
ON_CBN_SELCHANGE 142
ON_CBN_SELENDCANCEL 142
ON_CBN_SELENDOK 142
ON_CBN_SETFOCUS 142
ON_EN_CHANGE 115

ON_EN_ERRSPACE 115
ON_EN_HSCROLL 115
ON_EN_KILLFOCUS 116
ON_EN_MAXTEXT 116
ON_EN_SETFOCUS 116
ON_EN_UPDATE 116
ON_EN_VSCROLL 116
ON_LBN_DBLCLK 124
ON_LBN_ERRSPACE 125
ON_LBN_KILLFOCUS 125
ON_LBN_SELCANCEL 124
ON_LBN_SELCHANGE 125
ON_LBN_SETFOCUS 125
ON_WM_CHARTOITEM 125
ON_WM_VKEYTOITEM 125
OPAQUE 21
OUT_CHARACTER_PRECIS 62
OUT_DEFAULT_PRECIS 62
OUT_DEVICE_PRECIS 62
OUT_OUTLINE_PRECIS 62
OUT_RASTER_PRECIS 62
OUT_STRING_PRECIS 62
OUT_STROKE_PRECIS 62
OUT_TT_ONLY_PRECIS 62
OUT_TT_PRECIS 62
OWNERREDRAW. 160

P

PATCOPY 36
PATINVERT 36
PATPAINT 36
PROOF_QUALITY 63
PS_ALTERNATE 54
PS_COSMETIC 54, 57
PS_DASH 53
PS_DASHDOT 53
PS_DASHDOTDOT 53
PS_DOT 53
PS_ENDCAP_FLAT 55
PS_ENDCAP_ROUND 55
PS_ENDCAP_SQUARE 55
PS_GEOMETRIC 54, 57
PS_INSIDEFRAME 53
PS_JOIN_BEVEL 55
PS_JOIN_MITER 55
PS_JOIN_ROUND 55
PS_NULL 53
PS_SOLID 53
PS_USERSTYLE 54, 57

R

R2_BLACK 22
R2_COPYPEN 22
R2_MASKNOTPEN 22
R2_MASKPEN 23
R2_MASKPENNOT 22
R2_MERGENOTPEN 22
R2_MERGEPEN 23
R2_MERGEPENNOT 22
R2_NOP 22
R2_NOT 22
R2_NOTCOPYPEN 22
R2_NOTMASKPEN 23
R2_NOTMERGEPEN 23
R2_NOTXORPEN 23
R2_WHITE 22
R2_XORPEN 23
RegisterHotKey 192
RGN_AND 35, 76
RGN_COPY 35, 76
RGN_DIFF 35, 76
RGN_OR 35, 76
RGN_XOR 35, 76
RUSSIAN_CHARSET 45, 61

S

SB_BOTTOM 150, 326
SB_ENDSCROLL 149, 150
SB_LEFT 149
SB_LINEDOWN 151, 326
SB_LINELEFT 149
SB_LINERIGHT 150
SB_LINEUP 151, 326
SB_PAGEDOWN 151, 326
SB_PAGELEFT 150
SB_PAGERIGHT 150
SB_PAGEUP 151, 326
SB_RIGHT 150
SB_THUMBPOSITION 150, 151
SB_THUMBTRACK 150, 151, 326
SB_TOP 151, 326
SBS_BOTTOMALIGN 151
SBS_HORZ 151
SBS_LEFTALIGN 152
SBS_SIZEBOX 152
SBS_SIZEBOXBOTTOMRIGHTALIGN 152
SBS_SIZEBOXTOPLEFTALIGN 152
SBS_TOPALIGN 152
SBS_VERT 152
SC_CLOSE 388
SC_HOTKEY 388
SC_HSCROLL 388
SC_ICON 388
SC_KEYMENU 388
SC_MAXIMIZE 388
SC_MINIMIZE 388
SC_MOUSEMOVE 389
SC_MOVE 389
SC_NEXTWINDOW 389

SC_PREVWINDOW 389
SC_RESTORE 389
SC_SCREENSAVE 389
SC_SIZE 389
SC_TASKLIST 389
SC_VSCROLL 389
SC_ZOOM 388
SCROLLINFO 155, 156
SetGraphicsMode 61
SetWindowLong 242
SHIFTJIS_CHARSET 45, 61
SIF_ALL 155
SIF_DISABLENOSCROLL 155
SIF_PAGE 155
SIF_POS 156
SIF_RANGE 156
splitBar 346
splitBorder 347
splitBox 346
splitIntersection 347
SRCAND 37
SRCCOPY 37
SRCERASE 37
SRCINVERT 37
SRCPAINT 37
SS_BITMAP 102
SS_BLACKFRAME 101
SS_BLACKRECT 101
SS_CENTER 101
SS_CENTERIMAGE 102
SS_ENHMETAFILE 102
SS_GRAYFRAME 101
SS_GRAYRECT 101
SS_ICON 101, 102
SS_LEFT 101
SS_LEFTNOWORDWRAP 102
SS_NOPREFIX 102
SS_RIGHT 102
SS_SIMPLE 102
SS_USERITEM 102
SS_WHITEFRAME 102
SS_WHITERECT 102
STRETCH_ANDSCANS 23
STRETCH_HALFTONE 23
STRETCH_ORSCANS 23
SYMBOL_CHARSET 45, 61
SYSTEM_FIXED_FONT 21
SYSTEM_FONT 20

T

TA_UPDATECP 41, 42
TB_BOTTOM 206
TB_ENDTRACK 206
TB_LINEDOWN 206
TB_LINEUP 206
TB_PAGEDOWN 206
TB_PAGEUP 206
TB_THUMBPOSITION 206
TB_THUMBTRACK 206
TB_TOP 206
TBS_AUTOTICKS 207
TBS_BOTH 207
TBS_BOTTOM 207
TBS_ENABLESELRANGE 207
TBS_HORZ 206
TBS_LEFT 207
TBS_NOTICKS 207
TBS_RIGHT 207
TBS_TOP 207
TBS_VERT 206
TEXTMETRIC 43
THAI_CHARSET 45, 61
TMP_CENTERALIGN 385
TMP_LEFTALIGN 385
TMP_LEFTBUTTON 385
TMP_RIGHTALIGN 385
TMP_RIGHTBUTTON 385
TMPF_DEVICE 45
TMPF_FIXED_PITCH 45
TMPF_TRUETYPE 45
TMPF_VECTOR 45
TRANSPARENT 21
TTN_NEEDTEXT 216
TTS_ALWAYSTIP 220
TTS_NOPREFIX 220
TURKISH_CHARSET 45, 61
TV_HITTESTINFO 277
TV_INSERTSTRUCT 270
TV_ITEM 270
TV_SORTCB 278
TVE_COLLAPSE 275
TVE_COLLAPSERESET 275
TVE_EXPAND 275
TVE_TOGGLE 275
TVGN_CARET 275, 276
TVGN_CHILD 276
TVGN_DROPHILITE 275, 276
TVGN_FIRSTVISIBLE 275, 276
TVGN_NEXT 276
TVGN_NEXTVISIBLE 276
TVGN_PARENT 276
TVGN_PREVIOUS 276
TVGN_PREVIOUSVISIBLE 276
TVGN_ROOT 276
TVHT_ABOVE 277
TVHT_BELOW 278
TVHT_NOWHERE 278
TVHT_ONITEM 278
TVHT_ONITEMBUTTON 278
TVHT_ONITEMICON 278
TVHT_ONITEMINDENT 278
TVHT_ONITEMRIGHT 278
TVHT_ONITEMSTATEICON 278

TVHT_TOLEFT 278
TVHT_TORIGHT 278
TVI_FIRST 269, 270
TVI_LAST 270
TVI_SORT 269, 270
TVIF_CHILDREN 271
TVIF_HANDLE 271
TVIF_IMAGE 271
TVIF_PARAM 271
TVIF_SELECTEDIMAGE 271
TVIF_STATE 271
TVIF_TEXT 271
TVIS_CUT 271
TVIS_DISABLED 271
TVIS_DROPHILITED 271
TVIS_EXPANDED 271
TVIS_EXPANDEDONCE 271
TVIS_FOCUSED 271
TVIS_OVERLAYMASK 271
TVIS_SELECTED 271
TVIS_STATEIMAGEMASK 272
TVN_BEGINDRAG 279
TVN_BEGINLABELEDIT 279
TVN_BEGINRDRAG 279
TVN_DELETEITEM 279
TVN_ENDLABELEDIT 279
TVN_GETDISPINFO 279
TVN_ITEMEXPANDED 279
TVN_ITEMEXPANDING 279
TVN_KEYDOWN 279
TVN_SELCHANGED 279
TVN_SELCHANGING 279
TVN_SETDISPINFO 279
TVS_DISABLEDRAGDROP 267
TVS_EDITLABELS 267
TVS_HASBUTTONS 267
TVS_HASLINES 267
TVS_LINESATROOT 267
TVS_SHOWSELALWAYS 267
TVSIL_NORMAL 268
TVSIL_STATE 268

U

UD_MAXVAL 214
UD_MINVAL 214
UDACCEL 215
UDN_DELTAPOS 210
UnregisterHotKey 193

V

VARIABLE_PITCH 64
VK_DOWN 206
VK_END 206
VK_HOME 206
VK_LEFT 206
VK_NEXT 206
VK_PRIOR 206
VK_RIGHT 206
VK_UP 206

W

WA_ACTIVE 325
WA_CLICKACTIVE 326
WA_INACTIVE 325
WHITE_BRUSH 20
WHITE_PEN 20
WHITENESS 36
WHITEONBLACK 23
WINDING 22
WM_HSCROLL 211
WM_KEYUP 206
WM_LBUTTONUP 206
WM_SETHOTKEY 192
WM_VSCROLL 211
WS_GROUP 111
WS_TABSTOP 111

Содержание

Введение **1**

ЧАСТЬ VII. ГРАФИЧЕСКИЙ ИНТЕРФЕЙС УСТРОЙСТВ (GDI) **5**

Глава 16. Классы графического интерфейса **7**

Контексты устройств 8
Графические объекты 10

Глава 17. Класс CDC **14**

Создание объекта класса и его связь с контекстом устройства 14
Данные класса 15
Конструктор 16
Инициализация 16
Доступ к объектам рисования 18
Функции контекста устройства 18
Настройка процесса рисования 18
Функции средств рисования 19
Установка объектов рисования 19
Функции настройки цветов и палитр 21
Режимы рисования 21
Режим отображения 24
Настройка режимов отображения 24
Преобразование координат 30
Функции рисования 30
Отображение регионов 30
Отсечения 30
Рисование линий 30
Базовые функции рисования 31
Отображение эллипсов и многоугольников 33
Контуры 33
Отображение битовых массивов 36
Функции, использующие битовые массивы 36
Прокрутка 39
Управление выводом текста 39
Вывод текста 39
Информация о шрифтах 45

Дополнительные функции ... 46
Интерфейс низкого уровня с устройствами ... 46
Управление процессом печати документов ... 46
Метафайлы ... 46

Глава 18. Классы графических объектов ... 47

Класс CGdiObject ... 47
Данные класса ... 47
Функции-члены класса ... 47
Карандаш (класс CPen) ... 52
Кисть (класс CBrush) ... 58
Шрифт (класс CFont) ... 59
Битовый массив (класс CBitmap) ... 66
Регион (класс CRgn) ... 74
Создание регионов ... 74
Комбинирование регионов ... 76

Глава 19. Пример приложения PQClock ... 79

Класс приложения ... 79
Класс окна ... 82

ЧАСТЬ VIII. СТАНДАРТНЫЕ ЭЛЕМЕНТЫ УПРАВЛЕНИЯ (CONTROLS) ... 95

Глава 20. Обзор стандартных элементов управления ... 97

Создание элементов управления ... 97
Использование редактора ресурсов для создания элементов управления ... 98
Создание элементов управления в тексте программы ... 98

Глава 21. Классы стандартных элементов управления ... 100

Статические элементы управления (класс CStatic) ... 100
Конструктор ... 100
Инициализация ... 100
Вывод битового массива ... 103
Вывод пиктограммы ... 104
Вывод курсора ... 105
Вывод расширенных метафайлов ... 105
Обработка сообщения WM_CTLCOLOR ... 106
Кнопки (класс CButton) ... 108
Инициализация ... 111
Функции для определения и изменения текущего состояния кнопок ... 112
Функции для определения и изменения стиля кнопки ... 113
Назначение изображений ... 113
Переопределяемые функции ... 114
Простейший текстовый редактор (класс CEdit) ... 115
Инициализация ... 116
Выделение текста ... 118
Работа со строками ... 119
Работа с блоками теста ... 121

Параметры и флаги ... 122
Функции только для Windows 95 ... 123
Список (класс CListBox) ... 124
Инициализация ... 125
Изменение параметров списка ... 127
Функции для работы с содержимым списка ... 129
Операции над элементами списка ... 132
Переопределяемые функции ... 134
Пример самоотображаемого списка ... 137
Комбинированный список (класс CComboBox) ... 141
Инициализация ... 142
Основные операции с комбинированным списком ... 144
Операции со строками и виртуальные функции ... 147
Полоса прокрутки (класс CScrollBar) ... 148
Создание объекта ... 151
Функции для работы с полосой прокрутки ... 153

Глава 22. Классы с расширенными возможностями ... 157

Кнопки с нанесенным битовым изображением (класс CBitmapButton) ... 157
Кнопки в блоке диалога ... 158
Кнопки, созданные программным путем ... 159
Функции-члены класса ... 159
Список, имеющий кнопки выбора (класс CCheckListBox) ... 161
Инициализация ... 161
Функции для работы со списком ... 161
Виртуальные функции ... 162
Список с возможностью перемещения элементов (класс CDragListBox) ... 170
Ограничения на использование ... 170
Создание списка ... 170
Функции, предназначенные для перемещения элементов ... 171
Методика обхода ограничений ... 172
Реализация копирования элементов списка ... 172

ЧАСТЬ IX. ОБЩИЕ ЭЛЕМЕНТЫ УПРАВЛЕНИЯ (COMMON CONTROLS) ... 177

Глава 23. Использование общих элементов управления ... 179

Обзор ... 179
Общие элементы управления в блоках диалога ... 180
Общие элементы управления в качестве дочерних окон ... 181
Уведомления от общих элементов управления ... 181

Глава 24. Вспомогательные элементы управления ... 183

Просмотр видео-клипов (класс CAnimateCtrl) ... 183
Применение элемента анимации ... 184
Использование класса CAnimateCtrl ... 185
Уведомления ... 186
Создание видео-клипов ... 188

Клавиша вызова (класс CHotKeyCtrl) ... 189
Глобальные клавиши вызова ... 192
Поточно-ориентированные клавиши вызова ... 192
Пример использования класса CHotKeyCtrl ... 193
Индикатор (класс CProgressCtrl) ... 199
Использование индикатора в панели диалога ... 201
Линейка с бегунком (класс CSliderCtrl) ... 205
Уведомления ... 205
Создание элемента управления ... 206
Параметры элемента управления ... 207
Положение меток ... 208
Положение бегунка ... 209
Спин (класс CSpinButtonCtrl) ... 209
Уведомления ... 210
Автоматическое изменение ... 212
Создание спина ... 212
Параметры элемента управления ... 213
Всплывающая подсказка (класс CToolTipCtrl) ... 215
Оконная поддержка ... 216
Панели инструментов с подсказками ... 218
Возвращаемся к классу CToolTipCtrl ... 219

Глава 25. Комплексные элементы управления ... 223

Список изображений (класс CImageList) ... 223
Создание списка ... 224
Изменение содержимого списка ... 225
Рисование изображений ... 226
Перемещение изображений ... 227
Информация о списке ... 231
Заголовок (класс CHeaderCtrl) ... 231
Создание элемента заголовка ... 232
Изменение элементов заголовка ... 234
Обработка извещений элемента заголовка ... 235
Элемент заголовка, внешний вид которого определяется пользователем ... 238
Просмотр списка (класс CListCtrl) ... 239
Режимы вывода ... 239
Создание просмотра списка ... 240
Параметры просмотра списка ... 242
Функции для работы со списком в целом ... 242
Списки изображений ... 244
Основные и дополнительные поля ... 245
Записи по запросу (Callback Items) ... 245
Изменение содержимого просмотра списка ... 246
Поиск и сортировка записей ... 250
Работа со столбцами ... 254
Редактирование надписей записей ... 255
Обработка уведомлений ... 256
Реализация просмотра списка с возможностью перемещения записей ... 257
Переопределяемые функции ... 258
ListCtrl — пример приложения ... 258

Просмотр дерева (класс CTreeCtrl) ... 266
Создание просмотра дерева ... 267
Списки изображений ... 268
Функции для работы с просмотром дерева в целом ... 268
Изменения содержимого дерева ... 269
Поиск и сортировка записей дерева ... 275
Обработка уведомлений ... 279

ЧАСТЬ X. АРХИТЕКТУРА "ДОКУМЕНТ/ПРЕДСТАВЛЕНИЕ" ... 281

Глава 26. Основные положения ... 283

Место объекта-приложения в архитектуре "документ/представление" ... 286
Класс CDocTemplate ... 286
Класс CSingleDocTemplate ... 291
Класс CMultiDocTemplate ... 292
Роль фреймов в архитектуре "документ/представление" ... 300

Глава 27. Документ и его представления ... 306

Документы ... 307
Класс CDocument ... 309
Класс CArchive ... 317
Представления ... 320
Класс CView ... 323
Класс CCtrlView ... 327
Класс CEditView ... 328
Классы CListView и CTreeView ... 334
Класс CScrollView ... 334
Класс CSplitterWnd ... 339

Глава 28. Печать и предварительный просмотр документов ... 349

Выбор и настройка параметров принтера ... 351
Создание контекста устройства ... 354
Печать документов и библиотека MFC ... 354
Предварительный просмотр документа ... 364

ЧАСТЬ XI. ЭЛЕМЕНТЫ ИНТЕРФЕЙСА ПОЛЬЗОВАТЕЛЯ ... 373

Глава 29. Меню ... 375

Основные типы меню ... 375
Класс CMenu ... 378
Создание меню на основе шаблона ... 386
Динамическое создание меню (приложение DynMenu) ... 391
Динамическое изменение меню ... 393
Системное меню ... 399
Контекстное меню ... 402
Создание собственных маркеров состояния ... 405
Самоотображение элементов меню ... 406

Глава 30. Акселераторы, графические и другие ресурсы 408

Акселераторы 408
Графические ресурсы 410
Строковые ресурсы 411
Ресурсы, определяемые программистом 412

ПРИЛОЖЕНИЯ 413

Приложение 1. Иерархия классов MFC 4.1 415

Приложение 2. Функции-члены класса CDC 418

Конструктор 418
Инициализация 418
Работа с контекстом устройства 419
Объекты рисования 420
Работа с цветом и цветовыми палитрами 420
Настройка параметров рисования 421
Настройка режимов отображения 421
Преобразование координат 423
Работа с регионами 423
Работа с областью отсечения 424
Рисование линий 425
Простейшее рисование изображений 426
Рисование эллипсов и многоугольников 427
Рисование изображений 428
Отображение текстовых строк 430
Работа со шрифтами 432
Управление устройствами 433
Прокрутка 434
Работа с метафайлами 434
Работа с контурами 434

Приложение 3. Значения параметра функции CDC::GetDeviceCaps 436

Предметный указатель 441