№ 12 Библиотека классов и обработка сообщений MFC

При разработке программ использовать библиотеку классов MFC

1,8,15,22	Написать приложение, рисующее в окне различные графические фигуры(окружность, эллипс, прямоугольник, квадрат, сектор, сегмент) в ответ на нажатия клавиш «1», «2», «3», «4», «5», «6» соответственно. Каждую фигуру закрашивать своим цветом. Добавить обработку нажатия на левую клавишу мыши, при котором необходимо стереть и обновить все окно. При нажатии на правую клавишу мыши - вывести 5 раз с разными координатами (можно с одинаковым приращением) строку текста (с определенным размером, цветом, начертанием символов) через каждые 3 сек.
2,9,16,23	Написать приложение, выводящее в центре окна текст различным шрифтом(меняется имя шрифта, размер, цвет, начертание символов) в ответ на нажатия клавиш «1», «2», … и т.д. (всего 9 различных стилей). Добавить обработку нажатия на правую клавишу мыши, при котором будут рисоваться окружности с разным цветом, радиусом и координатами. Двойное нажатие на правую клашиу мыши прекращает рисование и очищает окно.
3,10,17,24	Написать приложение, рисующее в окне различные графические фигуры(окружность, эллипс, прямоугольник, квадрат, сектор, сегмент) с поясняющей надписью (название фигуры). Фигуры выводятся по сигналу от таймера с заданной частотой. По каждому следующему сигналу таймера изображение на экране(фигуры) меняется. Добавить обработку двойного нажатия на левую клавишу мыши увеличивает окно и продолжает вывод фигур в соответствии с алгоритмом. При достижении максимального размера окна оно уменьшается.
4,11,18,25	Написать приложение, рисующее в окне различные

	графические фигуры(окружность, эллипс, прямоугольник, квадрат, сектор, сегмент) в ответ на нажатия клавиши мыши по циклу. Добавить обработку нажатия на левую клавиши мыши, при котором будет выводится текущее время (менять шрифт, размер и цвет текста) GetCurrentTime().
5,12,19,26	Написать приложение, выводящее в центре окна текст различным шрифтом(меняется имя шрифта, размер, цвет, начертание символов) в ответ на нажатия клавиши мыши (всего 9 различных стилей). Добавить обработку нажатия на клавишу клавиатуры, при которой выводятся прямоугольники слева направо, сверху вниз.
6,13,20	Написать приложение, рисующее в окне окружности разного радиуса и цвета в ответ на нажатия клавиш «1», «2», «3», «4», «5», «6» соответственно. Добавить обработку нажатия на левую клавшиу мыши, при которой выводится текст 10 раз с увеличивающимся размером через каждые 2 сек.
7,14,21	Написать приложение, рисующее в ответ на нажатия клавиш «1», «2», «3», «4», «5», «6» соответствующие цифры с различным размером и цветом. Добавить обработку одинарного и двойного нажатия на правую и левую клавиши мыши, при которых выводятся эллипс, прямоугольник, квадрат, сектор соответственно.

Теория

Имена, используемые в MFC

Библиотека MFC содержит большое количество классов, структур, констант и т.д. Для того, чтобы текст MFC-приложений был более легким для

понимания, принято применять ряд соглашений для используемых имен и комментариев.

Названия всех классов и шаблонов классов библиотеки MFC начинаются с заглавной буквы С. При наследовании классов от классов MFC можно давать им любые имена. Рекомендуется начинать их названия с заглавной буквы С (class). Это сделает исходный текст приложения более ясным для понимания.

Чтобы отличить элементы данных, входящих в класс, от простых переменных, их имена принято начинать с префикса m_ (member – элемент или член класса). Названия методов классов, как правило, специально не выделяются, но обычно их начинают с заглавной буквы.

Библиотека MFC включает в себя, помимо классов, набор служебных функций или глобальных. Названия этих функций начинаются с символов Afx, например AfxGetApp. Символы AFX являются сокращением от словосочетания Application FrameworkX, означающих основу приложения, его внутреннее устройство.Например, очень часто используется функция AfxMessageBox(), отображающая заранее определенное окно сообщения. Но есть и член-функция MessageBox(). Таким образом, часто глобальные функции перекрываются функциями-членами.

Символы AFX встречаются не только в названии функций MFC. Многие константы, макрокоманды и другие символы начинаются с этих символов. В общем случае AFX является признаком, по которому можно определить принадлежность того или иного объекта (функция, переменная, ключевое слово или символ) к библиотеке MFC.

Функции-члены в MFC

Большинство функций, вызываемых в MFC-программе, являются членами одного из классов, определенных в библиотеке. Большинство функций API доступны через функции-члены MFC. Тем не менее, всегда можно обращаться к функциям API напрямую. Иногда это бывает необходимым, но все же в большинстве случаев удобнее использовать функции-члены MFC.

Иерархия классов MFC

Библиотека MFC содержит большую иерархию классов, написанных на C++. Структура иерархии приведена на рис. В ее вершине находится класс CObject, который содержит различные функции, используемые во время выполнения программы и предназначенные, в частности, для предоставления информации о текущем типе во время выполнения, для диагностики, и для сериализации. На вершине иерархии находится единственный базовй класс

CObject. Все остальные классы MFC можно условно разделить в зависимости от их отношения к CObject на производные и непроизводные. Для того чтобы составить представления о структуре и возможностях библиотеки приведем краткое описание классов.

Самый базовый класс библиотека MFC (класс CObject). Методы и элементы данных класса CObject представляют наиболее общие свойства наследованных из него классов MFC. Основное назначение класса CObject: хранение информации о классе времени выполнения; поддержка сериализации и диагностики объекта.

Единственной перменной этого класса является статическая переменная ClassObject типа CRuntimeClass, которая хранит информацию об объектк времни выполнения, ассоциированным с классом CObject.

Сериализация

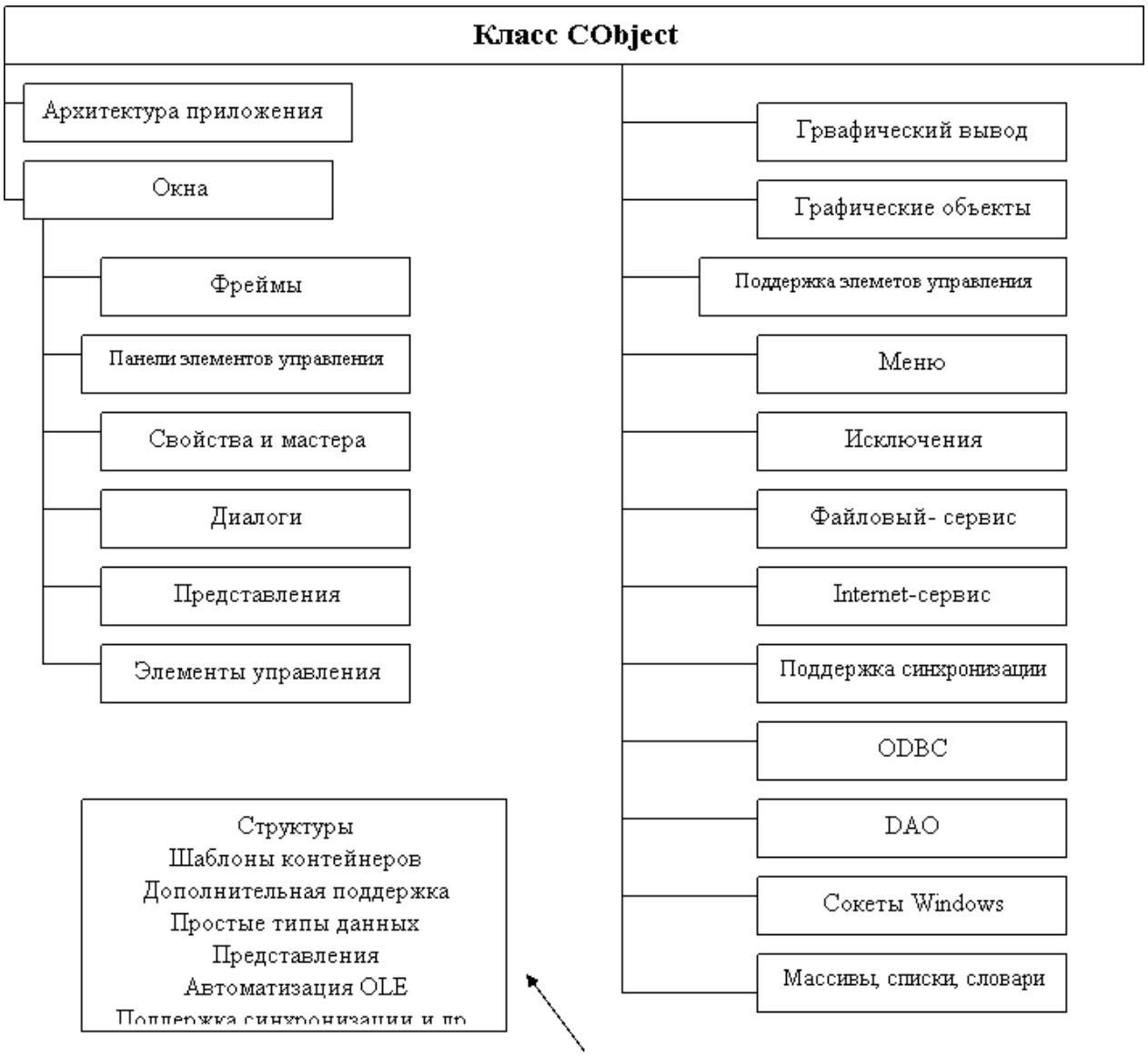
Сериализация - это механизм, позволяющий преобразовать текущее состояние объекта в последовательный поток байт, который обычно затем записывается на диск, и восстановить состояние объекта из последовательного потока, обычно при чтении с диска. Это позволяет сохранять текущее состояние приложения на диске, и восстанавливать его при последующем запуске. Этот метод можно разделит на функционально на две составляющие:

Наличие определенной виртуальной функции Serialize позволяет унифицировать процесс сохранения/восстановления объектов. Единственное что нужно – переопределить функцию.

С другой стороны, сериализация поддерживает механизм динамического создания объектов неизвестного заранее типа. Например приложение должно сохранять и восстанавливать некоторое количество объектов различного типа.

Диагностика

Каждый класс, производный от CObject, может по запросу проверить свое внутреннее состояние и выдать диагностическую информацию. Это интенсивно используется в MFC при отладке.



Классы не производные от CObject

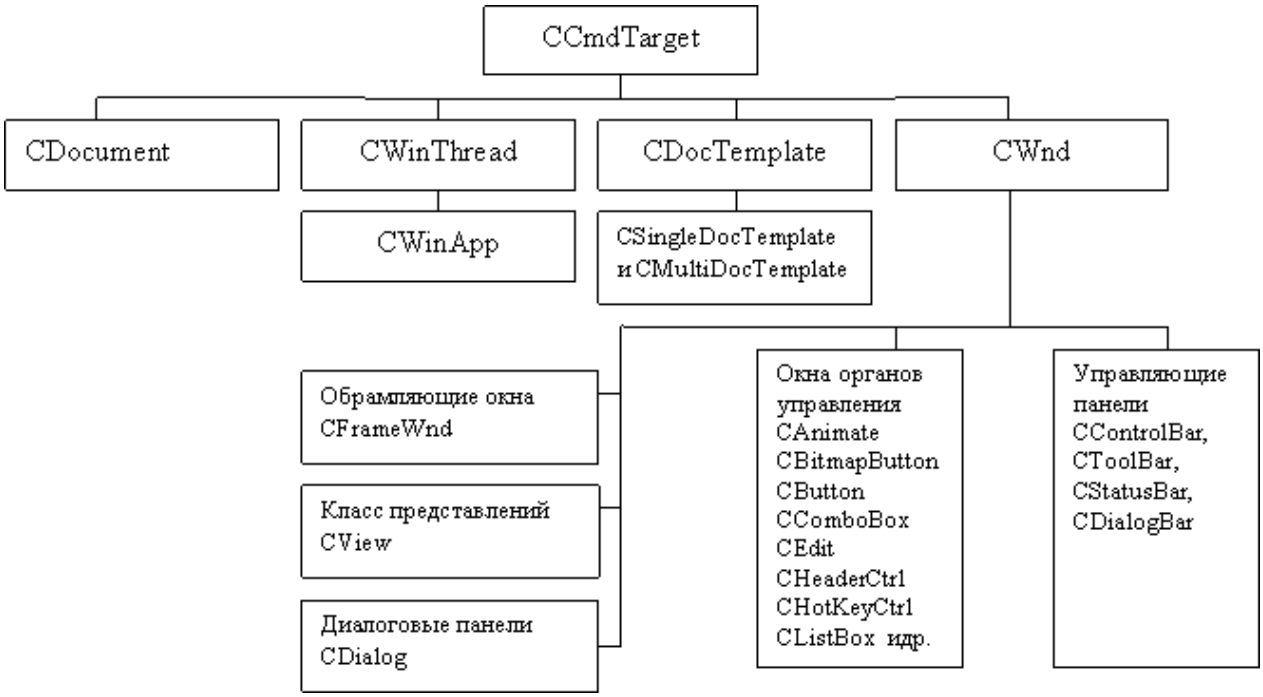
Классы, определяющие архитектуру приложения

Эти классы порождаются непосредственно от CObject. Класс CCmdTarget предназначен для обработки сообщений. Класс CFile предназначен для работы с файлами. Класс CDC обеспечивает поддержку контекстов устройств. Об контекстах устройств мы будем говорить несколько позднее. В этот класс включены практически все функции графики GDI. CGDIObject является базовым классом для различных DGI-объектов, таких как перья, кисти, шрифты и другие. Класс CMenu предназначен для манипуляций с меню. От класса CCmdTarget порождается очень важный класс CWnd. Он является базовым для создания всех типов окон, включая масштабируемые ("обычные") и диалоговые, а также различные элементы управления. Наиболее широко используемым производным классом является CFrameWnd. Как Вы увидите в

дальнейшем, в большинстве программ главное окно создается с помощью именно этого класса.

От класса CCmdTarget, через класс CWinThread, порождается, наверное, единственный из наиболее важных классов, обращение к которому в MFC-программах происходит напрямую: CWinApp. Это один из фундаментальных классов, поскольку предназначен для создания самого приложения. В каждой программе имеется один и только один объект этого класса. Как только он будет создан, приложение начнет выполняться.

Основа структуры приложения (класс CCmdTarget)



Подзадачи приложения (классы CWinThread и CWinApp)

От класса CCmdTarget наследуется класс CWinThread, представляющий подзадачи приложения. Простые приложения, которые будут рассматриваться дальше, имеют только одну подзадачу. Эта подзадача, называемая главной, представляется классом CWinApp, наследованным от класса CWinThread.

Документ приложения (класс CDocument)

Большинство приложений работают с данными или документами, хранимыми на диске в отдельных файлах. Класс CDocument, наследованный от базового класса CCmdTarget, служит для представления документов приложения.

Шаблон документов (классы CDocTemplate, CSingleDocTemplate и CMultiDocTemplate)

Еще один важный класс, наследуемы от CCmdTarget, называется CDocTemplate. От этого класса наследуется два класса: CSingleDocTemplate и CMultiDocTemplate. Все эти классы предназначены для синхронизации и управления основными объектами , представляющими приложение, - окнами, документами и используемыми ими ресурсами.

Окна (класс CWnd)

Практически все приложения имеют пользовательский интерфейс, построенный на основе окон. Это может быть диалоговая панель, одно окно или несколько окон, связанных вместе. Основные свойства окон представлены классом CWnd, наследованным от класса CCmdTarget.

Программисты очень редко создают объекты класса CWnd. Класс CWnd сам является базовым классом для большого количества классов, представляющих разнообразные окна. Перечислим классы, наследованные от базового класса CWnd.

Обрамляющие окна (класс CFrameWnd). Класс CFrameWnd представляет окна, выступающие в роли обрамляющих окон, в том числе главные окна приложений. От этого класса также наследуются классы CMDIChildWnd и CMDIFrameWnd, используемые для отображения окон многооконного интерфейса MDI. Класс CMDIFrameWnd представляет главное окно приложения MDI, а класс CMDIChildWnd - его дочерние окна MDI. Класс CMiniFrameWnd применяется для отображения окон уменьшенного размера. Такие окна обычно используются для отображения в них панели управления.

Окна органов управления

Для работы с органами управления (кнопки, полосы прокрутки, редакторы текста и т.д.) в библиотеке MFC предусмотрены специальные классы, наследованные непосредственно от класса CWnd. Перечислим эти классы:

- CAnimate - используется для отображения видеоинформации.
- CBitmapButton - кнопка с рисунком.
- CButton - кнопка.
- CComboBox - список с окном редактирования.
- CEdit - поле редактирования.
- CHeaderCtrl - заголовок для таблицы.
- CHotKeyCtrl - предназначен для ввода комбинации клавиш акселераторов.
- CListBox - список.
- CListCtrl - может использоваться для отображения списка пиктограмм.
- CProgressCtrl - линейный индикатор.
- CPropertySheet - блокнот. Может состоять из нескольких страниц.
- CRichEditControl - окно редактирования, в котором можно редактировать форматированный текст.
- CScrollBar - полоса просмотра.
- CSliderCtrl - движок.
- CSpinButtonCtrl - обычно используется для увеличения или уменьшения значения какого-либо параметра.
- CStatic - статический орган управления.
- CTabCtrl - набор "закладок".
- CToolBarCtrl - панель управления.
- CToolTipCtrl - маленькое окно, содержащее строку текста.
- CTreeCtrl - орган управления, который позволяет просматривать иерархические структуры данных.

Управляющие панели (классы CControlBar, CToolBar, CStatusBar, CDialogBar)

Класс CControlBar и классы, наследуемые от него, предназначены для создания управляющих панелей. Такие панели могут содержать различные органы управления и отображаются, как правило, в верхней или нижней части главного окна приложения.

Так, класс CToolBar предназначен для создания панели управления. Эта панель обычно содержит ряд кнопок, дублирующих действие меню приложения.

Класс CStatusBar управляет панелью состояния. Панель состояния отображается в виде полосы в нижней части окна. В ней приложение может отображать всевозможную информацию, например, краткую подсказку о выбранной строке меню.

Большие возможности представляет управляющая панель, созданная на основе класса CDialogBar. Такая панель использует обычный шаблон диалоговой панели, которую можно разработать в редакторе ресурсов Visual C++.

Окна просмотра (класс CView и классы, наследованные от него)

Большой интерес представляют класс CView и классы, наследуемые от него. Эти классы представляют окно просмотра документов приложения. Именно окно просмотра используется для вывода на экран документа, с которым работает приложение. Через это окно пользователь может изменять документ.

Разрабатывая приложение, программисты наследуют собственные классы просмотра документов либо от базового класса CView, либо от одного из нескольких порожденных классов, определенных в библиотеке MFC.

Классы, наследованные от CCtrlView, используют для отображения готовые органы управления. Например, класс CEditView использует орган управления edit (редактор).

Класс CScrollView представляет окно просмотра, которое имеет полосы свертки. В классе определены специальные методы, управляющие полосами просмотра.

Класс CFormView позволяет создать окно просмотра документа, основанное на диалоговой панели. От этого класса наследуется еще два класса CRecordView и CDaoRecordView. Эти классы используются для просмотра записей баз данных.

Диалоговые панели (класс CDialog и классы, наследованные от него)

От базового класса наследуются классы, управляющие диалоговыми панелями. Если необходимо создать диалоговую панель, можно наследовать класс от CDialog.

Вместе с диалоговыми панелями обычно используется класс CDataExchange. Класс CDataExchange обеспечивает работу процедур обмена данными DDX (Dialog Data Exchange) и проверки данных DDV (Dialog Data Validation), используемых для диалоговых панелей. В отличие от CDialog класс CDataExchange не наследуется от какого-либо другого класса.

От класса CDialog наследуется ряд классов, представляющих собой стандартные диалоговые панели для выбора шрифта, цвета, вывода документа на печать, поиска в документе определенной последовательности

символов, а также поиска и замены одной последовательности символов другой последовательностью.

Чтобы создать стандартный диалог, можно просто определить объект соответствующего класса. Дальнейшее управление такой панелью осуществляется методами класса.

Массивы, списки, словари

В состав MFC включен целый набор классов, предназначенных для хранения информации в массивах, списках и словарях. Все эти классы наследованы от базового класса CObject.

Несмотря на то, что в языке С определено понятие массива, классы MFC обеспечивают более широкие возможности. Например, можно динамически изменять размер массива, определенного с помощью соответствующего класса.

Для представления массивов предназначены следующие классы:

- CByteArray - байты.
- CDWordArray - двойные слова.
- CObArray - указатели на объекты класса CObject.
- CPtrArray - указатели типа void.
- CStringArray - объекты класса CString.
- CUIntArray - элементы класса unsigned integer или UINT.
- CWordArray - слова.

Для решения многих задач применяются такие структуры хранения данных, как списки. MFC включает ряд классов, наследованных от базового класса CObject, которые представляют программисту готовое для создания собственных списков. В этих классах определены все методы, необходимые при работе со списками, - добавление нового элемента, вставка нового элемента, определение следующего или предыдущего элемента в списке, удаление элемента и т.д.

Перечислим классы списков, которые позволяют построить списки из элементов любых типов любых классов:

CObList - указатели на объекты класса CObject.

CPtrList - указатели типа void.

CStringList - объекты класса CString.

В библиотеке MFC определена еще одна группа классов, позволяющая создавать словари. Словарь представляет собой таблицу из двух колонок, устанавливающих соответствие двух величин. Первая величина представляет ключевое значение и записывается в первую колонку, а вторая - связанное с ней значение, хранящееся во второй колонке. Словарь позволяет добавлять в

него пары связанных величин и осуществлять выборку значений по ключевому слову.

Для работы со словарями используются классы:

- CMapPtrToPtr - ключевое слово - указатель типа void, связанное с ним значение - указатель типа void.
- CMapPtrToWord - ключевое слово - указатель типа void, связанное с ним значение - слово.
- CMapStringToOb - ключевое слово - объекты класса CString, связанное с ним значение - указатель на объекты класса CObject.
- CMapStringToPtr - ключевое слово - объекты класса CString, связанное с ним значение - указатель типа void.
- CMapStringToString - ключевое слово - объекты класса CString, связанное с ним значение - на объекты класса CObject.
- CMapWordToOb - ключевое слово - слово, связанное с ним значение - указатель на объекты класса CObject.
- CMapWordToPtr- ключевое слово - слово, связанное с ним значение - указатель типа void.

Файловая система (класс CFile)

Библиотека MFC включает класс для работы с файловой системой компьютера. Он называется CFile и также наследуется от базового класса CObject. Непосредственно от класса CFile наследуется еще несколько классов - CMemFile, CStdioFile, CSocketFile.

При работе с файловой системой может потребоваться получить различную информацию о некотором файле, например, дату создания, размер и т.д. Для хранения этих данных предназначен специальный класс CFileStatus. Класс CFileStatus - один из немногих классов, которые не наследуются от базового класса CObject.

Контекст отображения (класс CDC)

Для отображения информации в окне или на любом другом устройстве приложение должно получать так называемы контекст отображения. Основные свойства контекста отображения определены в классе CDC. От него наследуется 4 различных класса, представляющие контекст отображения различных устройств.

Дадим краткое описание классов, наследованных от CDC:

CClientDC - контекст отображения, связанный с внутренней областью окна (client area). Для получения контекста конструктор класса вызывает функцию программного интерфейса GetDC, а деструктор - функцию ReleaseDC.

CMetaFileDC - класс предназначен для работы с метафайлами.

CPaintDC - конструктор класса CPaintDC для получения контекста отображения вызывает метод CWnd::BeginPaint, деструктор - метод CWnd::EndPaint. Объекты данного класса можно использовать только при обработке сообщения WM_PAINT. Это сообщение обычно обрабатывает метод OnPaint.

CWindowDC - контекст отображения, связанный со всем окном. Для получения контекста конструктор класса вызывает функцию программного интерфейса GetWindowDC, а деструктор - функцию ReleaseDC.

Объекты графического интерфейса (класс CGdiObject)

Для отображения информации используются различные объекты графического интерфейса - GDI-объекты. Для каждого из этих объектов библиотека MFC содержит описывающий его класс, наследованный от базового класса CGdiObject.

Для работы с GDI-объектами используются классы:

- CBitmap - растровое изображение bitmap.
- CBbrush - кисть.
- CFont - шрифт.
- CPalette - палитра цветов.
- CPen - -перо.
- CRgn - область внутри окна.

Меню (класс CMenu)

Практически каждое приложение имеет собственное меню. Оно, как правило, отображается в верхней части главного окна приложения. Для управления меню в состав MFC включен специальный класс CMenu, наследованный непосредственно от базового класса CObject.

Для управления меню и панелями используется также класс CCmdUI. Этот класс не наследуется от базового класса CObject.

Объекты класса CCmdUI создаются, когда пользователь выбирает строку меню или нажимает кнопки панели управления. Методы класса CCmdUI позволяют управлять строками меню и кнопками панели управления. Например, существует метод, который делает строку меню неактивной.

Другие классы

В MFC включено несколько классов, обеспечивающих поддержку приложений, работающих с базами данных. Это такие классы, как CDataBase, CRecordSet, CDaoDataBase. CDaoRecordSet, CDaoQueryDef, CDaoTableDef, CDaoWorkSpace, CLongBinary, CFieldExchange и CDaoFieldExchange.

Библиотека MFC позволяет создавать многозадачные приложения. Для синхронизации отдельных задач приложения предусмотрен ряд специальных классов. Все они наследуются от класса CSyncObject, представляющий собой абстрактный класс.

В некоторых случаях требуется, чтобы участок программного кода мог выполняться только одной задачей. Такой участок называют критической секцией кода. Для создания и управления критическими секциями предназначены объекты класса CCriticalSection.

Объекты класса CEvent представляют событие. При помощи событий одна задача приложения может передать сообщение другой.

Объекты класса CMutex позволяют в данный момент предоставить ресурс в пользование одной только задаче. Остальным задачам доступ к ресурсу запрещается.

Объекты класса CSemafore представляют собой семафоры. Семафоры позволяют ограничить количество задач, которые имеют доступ к какому-либо ресурсу.

Для программистов, занимающихся сетевыми коммуникациями, в состав библиотеки MFC включены классы CAsyncSocket и наследованный от него класс CSocket. Эти классы облегчают задачу программирования сетевых приложений.

Кроме уже описанных классов библиотека MFC включает большое количество классов, предназначенных для организации технологии OLE.

Классы, не имеющие базового класса

Кроме классов, наследованных от базового класса CObject, библиотека MFC включает ряд самостоятельных классов. У них нет общего базового класса, и имеют различное назначение.

Несколько классов, которые не наследуются от базового класса CObject, уже упоминались. К ним относятся классы CCmdUI, CFileStatus, CDataExchange, CFieldExchange и CDaoFieldExchange.

Простые классы. Библиотека MFC содержит классы, соответствующие объектам типа простых геометрических фигур, текстовых строк и объектам, определяющим дату и время:

- CPoint - объекты класса описывают точку.
- CRect - объекты класса описывают прямоугольник.
- CSize - объекты класса определяют размер прямоугольника.
- CString - объекты класса представляют собой текстовые строки переменной длины.
- CTime - объекты класса служат для хранения даты и времени. Большое количество методов класса позволяет выполнять над объектами класса различные преобразования.
- CTimeSpan - объекты класса определяют период времени.

Архивный класс (класс CArchive). Класс CArchive используется для сохранения и восстановления состояния объектов в файлах на диске. Перед использованием объекта класса CArchive он должен быть привязан к файлу - объекта класса CFile.

Информация о классе объекта (структура CRuntimeClass). Во многих случаях бывает необходимо уже во время работы приложения получать информацию о классе и его базовом классе. Для этого любой класс, наследованный от базового класса CObject, связан со структурой CRuntimeClass. Она позволяет определить имя класса объекта, размер объекта в байтах, указатель на конструктор класса, не имеющий аргументов, и деструктор класса. Можно также узнать подобную информацию о базовом классе и некоторые дополнительные сведения.

Отладка приложения (классы CDumpObject, CMemoryState). В отладочной версии приложения можно использовать класс CDumpContext. Он позволяет выдавать состояние различных объектов в текстовом виде.

Класс CMemoryState позволяет локализовать проблемы, связанные с динамическим выделением оперативной памяти. Такие проблемы обычно возникают, когда пользователь выделяет память, применяя оператор new, а затем забывает вернуть эту память операционной системе.

Печать документа (класс CPrintInfo). Класс CPrintInfo предназначен для управления печатью документов на принтере. Когда пользователь отправляет документ на печать или выполняет предварительный просмотр документа перед печатью, создается объект класса CPrintInfo. Он содержит различную информацию о том, какие страницы документа печатаются, и т.д.

Каркас MFC-программы

В простейшем случае программа, написанная с помощью MFC, содержит два класса, порождаемые от классов иерархии библиотеки: класс,

предназначенный для создания приложения, и класс, предназначенный для создания окна. Другими словами, для создания минимальной программы необходимо породить один класс от CWinApp, а другой - от CFrameWnd. Эти два класса обязательны для любой программы.

Класс приложение

В классе приложения необходимо:

- Переопределить виртуальную функцию InitInstance
- Создать внутри нее объект оконного класса

Например, стандартный шаблон класса приложение может выглядеть так :

```
#include <afxwin.h>
class CMyApp: public CWinApp
{ public:
        virtual BOOL InitInstance()
{
                m_pMainWnd = new CMyWindow;
                m_pMainWnd -> ShowWindow(SW_SHOWNORMAL);
                return TRUE;
}
};
CMyApp app;
                virtual BOOL CWinApp::InitInstance();
```

Это виртуальная функция, которая вызывается каждый раз при запуске программы. В ней должны производиться все действия, связанные с инициализацией приложения. Функция должна возвращать TRUE при успешном завершении и FALSE в противном случае. В нашем случае, в функции сначала создается объект класса CMyApp, и указатель на него запоминается в переменной m_pMainWnd. Эта переменная является членом класса CWinThread. Она имеет тип CWnd* и используется почти во всех MFC-программах, потому что содержит указатель на главное окно. В последующих двух строчках через нее вызываются функции-члены окна. Когда окно создано, вызывается функция с прототипом:

```
BOOL CWnd::ShowWindow(int How);
```

Параметр определяет, каким образом окно будет показано на экране.

Строка CMyApp app – это объявление глобального app класса приложение. Каждое приложение, созданное с помощью MFC, должно содержать только один объект класса, производный от CWinApp. В конструкторе базового класса CWinApp, который вызывается при создании объекта app запускается функция WinMain, которая, в свою очередь, запускает цикл ожидания и обработки сообщений, условно называемый MessagePump.

Оконный класс

Оконный класс удобно наследовать о класса CFrameWnd. Самая простая реализация оконного класса, в которой создается пустое окно с атрибутами по умолчанию, но “своим” заголовком будет:

```
class CMyWindow : public CFrameWnd{
public:
        CMyWindow()   // конструктор
        {
                Create(NULL,"Простой диалог");  // окно-рамка по умолч-
                                                        анию.
        }
};
```

Этот код необходимо вставить до определения класса CMyApp. Объект класса, производного от CFrameWnd, создаётся как окно-рамка. Термин “Окно-рамка” используется для обозначения тех элементов интерфейса окна, которые позволяют управлять его обликом: обрамление, заголовок, системное меню, кнопки минимизации и максимизации, восстановления окна. Вся остальная область окна называется клиентской областью. Именно в этой области располагаются элементы: меню, строи статуса, панель инструментов и другие отображаемые элементы. Существует три способа создания окна-рамки: явное конструирование путём вызова метода Create(как внашем примере), загрузка окна, атрибуты которого заданы в файле ресурсов путём вызова метода LoadFrame, неявное конструирование на основе шаблона-документа.

Для простейшей программы необходимо создать два объекта MFC:

1. объект класса CWinApp;
2. объект класса CWnd.

Вспомним, что при разработке простейшего приложения с использованием функции API, мы создавали две функции: WinMain и оконную процедуру WndProc. Существует аналогия между этой структурой и структурой простейшего приложения на основе MFC. Аналогом WinMain является класс приложения CWinApp, а аналогом оконной процедуры – класс окна CWnd. Мы должны создать свой класс, производный от CWinApp, например, CFirstApp.

CObject→CCmdTarget→CWinThread→CWinApp

Всю работу, которую выполняет функция WinMain, выполняет теперь класс CWinApp. Нам остаётся только:

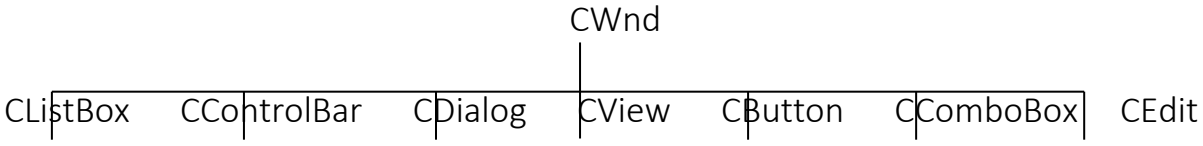
- переопределить в классе CFirstApp виртуальную функцию InitInstance;

- создать внутри этой функции объект CWnd – аналог оконной процедуры.

CObject→CCmdTarget→CWnd→CFrameWnd

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

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



Рассмотрим некоторые функции – члены класса CWinApp.

Конструктор CWinApp (LPCTSTR lpszAppName = NULL);

Параметр lpszAppName – указатель на строку, которая содержит имя приложения, используемого системой Windows. Если этот аргумент отсутствует или равен NULL, то для формирования имени приложения конструктор использует строку из файла ресурсов с идентификатором AFX_IDS_APP_TITLE, а при её отсутствии – имя исполняемого файла.

Функция virtual BOOL InitInstance();

Эта функция выполняет инициализацию каждого нового экземпляра приложения. В классе CWinApp эта функция ничего не делает и Вы должны переопределить её в производном классе. В этой функции создается объект “главное окно” приложения, загружаются стандартные настройки из INI-файла или из реестра Windows, создается новый или открывается существующий документ, регистрируются шаблоны документов, создаются сами документы, их представления (вид) и ассоциированные с ними окна.

virtual BOOL ExitInstance();

Функция вызывается только из функции Run для завершения работы приложения.

virtual int Run();

Функция запускает цикл обработки сообщений.

CFrameWnd – класс «окно - рамка».

Класс CFrameWnd служит для создания перекрывающихся или всплывающих окон и поддерживает однодокументный интерфейс Windows (SDI). Объект этого класса координирует взаимодействие приложения с документами и его представлением. Он отражается на экране в виде тех элементов интерфейса, которые позволяют управлять обликом окна: обрамление, строка заголовка, системное меню, кнопки минимизации, максимизации, восстановления. Это окно отвечает за управление размещением своих дочерних окон и других элементов рабочей (клиентской) области. Кроме того, это окно переадресует команды своими представлениями и может отвечать на извещения от элементов управления окна.

Существует три способа создания объекта CFrameWnd:

1. Явное конструирование путем вызова метода Create;
2. Загрузка окна, атрибуты которого заданы в файле ресурсов (rc - файле) путем вызова метода LoadFrame;
3. Неявное конструирование на основе шаблона документа.

Мы воспользовались первым способом и задали нуль в качестве первого аргумента функции Create. Это означает, что используется стиль оконного класса, заданный по умолчанию. Если вы хотите повлиять на процесс регистрации оконного класса, а тем самым и на стиль окна, то перед созданием окна следует вызвать функцию AfxRegisterWndClass и передать ей в качестве параметра нужные атрибуты окна.

Пример

```
//simpwin.h

#include <afxwin.h>
// Класс основного окна приложения
class CMainWin: public CFrameWnd {
                                public:
CMainWin();
                        // Декларирование карты сообщений
DECLARE_MESSAGE_MAP()
};
// Класс приложения. Должен существовать только
// один экземпляр этого класса.
// Член-функция InitInstance() вызывается при запуске
// приложения.
class CApp: public CWinApp {
public:
BOOL InitInstance();
};
```

```
/* simpwin.cpp*/

 #include <afxwin.h>
 #include <string.h>
 #include "SIMPWIN.H"
 // Создание одного и только одного экземпляра
// приложения
 CApp App;
 // Реализация
BOOL CApp::InitInstance()
 {
 // Создание главного окна приложения и его
// отображение.
 // Член CApp::m_pMainWnd - это указатель на объект
// главного окна.
 m_pMainWnd = new CMainWin;
 m_pMainWnd->ShowWindow(SW_RESTORE);
 m_pMainWnd->UpdateWindow();
 // Сигнализируем MFC об успешной инициализации
// приложения.
 return TRUE;
 }

 CMainWin::CMainWin()
 {
 // Создание окна с заголовком. Используется
// встроенный в MFC
 // класс окна, поэтому первый параметр 0.
 this->Create(0, L"Простейшее приложение на MFC");
 }
 // Реализация карты сообщений
 BEGIN_MESSAGE_MAP(CMainWin /*класс окна*/, CFrameWnd
/*класс-предок*/)
                        END_MESSAGE_MAP()
```

В примере была использована функция Create(), которая на самом деле имеет много параметров. Ее прототип таков:

```
BOOL CFrameWnd::Create(
LPCSTR ClassName,           // Имя Windows-класса окна
LPCSTR Title,               // Заголовок
DWORD Style = WS_OVERLAPPEDWINDOW, // Стиль
const RECT &XYSize = rectDefault, // Область
          CWnd *Parent = 0,          //Окно-предок
LPCSTR MenuName = 0,        //Имя ресурса меню
DWORD ExStyle = 0,          //Расширенные стили
CCreateContext *Context = 0  // Доп. данные
);
```

Первый параметр, ClassName, определяет имя класса окна для оконной подсистемы Windows. Обычно его не нужно явно задавать, так как MFC выполняет всю необходимую черновую работу. В данных методических указаниях мы не будем использовать своих классов окон. Параметр Style задает стиль окна. По умолчанию создается стандартное перекрываемое окно. Можно задать свой стиль, объединив с помощью операции "или" несколько констант.

Приложение без главного окна

```
//Файл first.cpp
    #include <afxwin.h> // Включаемый файл для MFC

    // Класс CFirstApp - главный класс приложения.
    // Наследуется от базового класса CWinApp.
    class CFirstApp:public CWinApp
    {
    public:
        // Переопределение метода InitInstance,
        // предназначенного для инициализациии приложения.
        virtual BOOL InitInstance();
    };

    // Создание объекта приложения класса CFirstApp.
    CFirstApp theApp;

    // Метод InitInstance
// Переопределение виртуального метода InitInstance класса CWinApp.
    // Он вызывается каждый раз при запуске приложения.
    BOOL CFirstApp::InitInstance()
    {
        AfxMessageBox("First MFC-application");
        return FALSE;
    }
```

Приложение с главным окном

```
//Файл start.h
    #include <afxwin.h>
    class CStartApp: public CWinApp
    {
        public:
        virtual BOOL InitInstance();
    };
```

```
//Файл startm.h
    #include <afxwin.h>
    // Класс CMainWindow - представляет главное окно приложения.
    class CMainWindow : public CFrameWnd
    {
    public:
        CMainWindow();
    };
//Файл start.cpp
    #include <afxwin.h>
    #include "start.h"
    #include "startm.h"
    BOOL CStartApp::InitInstance()
    {
        // Создание объекта класса CMainWindow
        m_pMainWnd= new CMainWindow();
        // Отображение окна на экране.
        // Параметр m_nCmdShow определяет режим отображения окна.
        m_pMainWnd->ShowWindow(m_nCmdShow);
        // Обновление содержимого окна.
        m_pMainWnd->UpdateWindow();
        return TRUE;
    }
    CStartApp theApp;
//Файл startm.cpp
    #include <afxwin.h>
    #include "startm.h"
    // Конструктор класса CMainWindow
    CMainWindow::CMainWindow()
    {
        // Создание окна приложения
        Create(NULL,"Hello");
    }
```

Обработка окном сообщений

Работа операционной системы Windows основана на обработке сообщений. Когда пользователь работает с устройствами ввода/вывода (например, клавиатурой или мышью), драйверы этих устройств создают сообщения, описывающие его действия.

Сообщения сначала попадают в системную очередь сообщений операционной системы. Из нее сообщения передаются приложениям, которым они предназначены, и записываются в очередь приложений. Каждое приложение имеет собственную очередь сообщений.

Приложение в цикле, который называется циклом обработки сообщений, получает сообщения из очереди приложения и направляет их соответствующей функции окна, которая и выполняет обработку сообщения. Цикл обработки сообщений в традиционной Windows-программе обычно состоял из оператора while, в котором циклически вызывались функции GetMessage и DispatchMessage. Для более сложных приложений цикл обработки сообщений содержал вызовы других функций (TranslateMessage, TranslateAccelerator). Они обеспечивали предварительную обработку сообщений.

Каждое окно приложения имеет собственную функцию окна. В процессе обработки сообщения операционная система вызывает функцию окна и передает ей структуру, описывающую очередное сообщение. Функция обработки сообщения проверяет, какое именно сообщение поступило для обработки, и выполняет соответствующие действия.

Если при создании приложения используется библиотека классов MFC, то за обработку сообщений отвечают классы. Любой класс, наследованный от базового класса CCmdTarget, может обрабатывать сообщения. Чтобы класс смог обрабатывать сообщения, необходимо, чтобы он имел таблицу сообщений класса. В этой таблице для каждого сообщения указан метод класса, предназначенный для его обработки.

Группы сообщений

Сообщения, которые могут обрабатываться приложением, построенным с использованием библиотеки классов MFC, делятся на 3 группы.

Оконные сообщения

Эта группа включает сообщения, предназначенные для обработки функцией окна. Практически все сообщения, идентификаторы которых начинаются префиксом WM_, за исключением WM_COMMAND,. относятся к этой группе.

Оконные сообщения предназначаются для обработки объектами, представляющими окна. Это могут быть практически любые объекты класса CWnd или классов, наследованных от него (CFrameWnd, CMDIFrameWnd, CMDIChildWnd, CView, CDialog). Характерной чертой этих классов является то, что они включают идентификатор окна.

Большинство этих сообщений имеют параметры, детально характеризующие сообщение.

Сообщения от органов управления

Эта группа включает в себя сообщения WM_COMMAND от дочерних окон (включая окна стандартных классов), передаваемых их родительскому окну.

Сообщения от органов управления обрабатываются точно таким же образом, что и оконные сообщения.

Исключение составляет сообщение WM_COMMAND с кодом извещения BN_CLICKED. Это сообщение передается кнопкой, когда пользователь на нее нажимает. Обработка сообщений с кодом извещения BN_CLICKED от органов управления происходит аналогично обработке командных сообщений.

Командные сообщения

Это сообщения WM_COMMAND от меню, кнопок панели управления и клавиш акселераторов. В отличие от оконных сообщений и сообщений от органов управления командные сообщения могут быть обработаны более широким спектром объектов. Эти сообщения обрабатывают не только объекты, представляющие окна, но также объекты классов, представляющих приложение, документы или шаблон документов.

Характерной особенностью командных сообщений является идентификатор. Идентификатор командного сообщения определяет объект, который вырабатывает (посылает) данное сообщение.

Таблица сообщений

В библиотеке классов MFC для обработки сообщений используется специальный механизм, который имеет название Message Map - таблица сообщений.

Таблица сообщений состоит из набора специальных макрокоманд, ограниченных макрокомандами BEGIN_MESSAGE_MAP и END_MESSAGE_MAP. Между ними расположены макрокоманды, отвечающие за обработку отдельных сообщений:

```
BEGIN_MESSAGE_MAP(ИмяКласса,ИмяБазовогоКласса)
        // макросы
    END_MESSAGE_MAP()
```

Макрокоманда BEGIN_MESSAGE_MAP представляет собой заголовок таблицы сообщений. Она имеет два параметра. Первый параметр содержит имя класса таблицы сообщений. Второй - указывает его базовый класс.

Если в таблице сообщений класса отсутствует обработчик для сообщения, оно передается для обработки базовому классу, указанному вторым параметром макрокоманды BEGIN_MESSAGE_MAP. Если таблица сообщений базового класса также не содержит обработчик этого сообщения, оно передается следующему базовому классу и т.д. Если ни один из базовых

классов не может обработать сообщение, выполняется обработка по умолчанию, зависящая от типа сообщения:

- стандартные сообщения Windows обрабатываются функцией обработки по умолчанию;
- командные сообщения передаются по цепочке следующему объекту, который может обработать командное сообщение.

Можно определить таблицу сообщений класса вручную, однако более удобно воспользоваться для этой цели средствами ClassWizard. ClassWizard не только позволяет в удобной форме выбрать сообщения, которые должен обрабатывать класс. Он включит в состав класса соответствующие методы-обработчики. Программисту останется только вставить в них необходимый код. К сожалению, использовать все возможности ClassWizard можно только в том случае, если приложение создано с применением средств автоматизированного программирования MFC AppWizard.

Рассмотрим макрокоманды, отвечающие за обработку различных типов сообщений.

Макрокоманда ON_WM_<name>. Обрабатывает стандартные сообщения операционной системы Windows. Вместо <name> указывается имя сообщения без префикса WM_. Например:

```
ON_WM_CREATE()
```

Для обработки сообщений, определенных в таблице макрокомандой On_WM_<name>, вызываются одноименные методы. Имя метода обработчика соответствует названию сообщения, без учета префикса WM_. В классе CWnd определены обработчики для стандартных сообщений. Эти обработчики будут использоваться по умолчанию.

Макрокоманды ON_WM_<name> не имеют параметров. Однако методы, которые вызываются для обработки соответствующих сообщений, имеют параметры, которые, количество и назначение которых зависит от обрабатываемого сообщения.

Если определить обработчик стандартного сообщения Window в своем классе, то он будет вызываться вместо обработчика, определенного в классе CWnd (или другом базовом классе). В любом случае можно вызвать метод-обработчик базового класса из своего метода-обработчика.

Макрокоманда ON_REGISTERED_MESSAGE. Эта макрокоманда обслуживает сообщения операционной системы Windows, зарегистрированные с помощью функции RegisterWindowMessage. Параметра nMessageVariable этой макрокоманды указывает идентификатор сообщения, для которого будет вызываться метод memberFxn:

```
ON_REGISTERED_MESSAGE(nMessageVariable, memberFxn)
```

Макрокоманда ON_MESSAGE. Данная макрокоманда обрабатывает сообщения, определенные пользователем. Идентификатор сообщения (его имя) указывается параметром message. Метод, который вызывается для обработки сообщения, указывается параметром memberFxn:

ON_MESSAGE(message,memberFxn)

Макрокоманда ON_COMMAND. Эти макрокоманды предназначены для обработки командных сообщений. Командные сообщения поступают от меню, кнопок панели управления и клавиш акселераторов. Характерной особенностью командных сообщений является то, что с ними связан идентификатор сообщения.

Макрокоманда ON_COMMAND имеет два параметра. Первый параметр соответствует идентификатору командного сообщения, а второй - имени метода, предназначенного для обработки этого сообщения. Таблица сообщений должна содержать не больше одной макрокоманды для командного сообщения:

ON_COMMAND(id,memberFxn)

Обычно командные сообщения не имеют обработчиков, используемых по умолчанию. Существует только небольшая группа стандартных командных сообщений, имеющих методы-обработчики, вызываемые по умолчанию. Эти сообщения соответствуют стандартным строкам меню приложение. Так например, если строке Open меню File присвоить идентификатор ID_FILE_OPEN, то для его обработки будет вызван метод OnFileOpen, определенный в классе CWinApp.

Макрокоманда ON_COMMAND_RANGE. Макрокоманда ON_COMMAND ставит в соответствие одному командному сообщению один метод-обработчик В некоторых случаях более удобно, когда один и тот же метод-обработчик применяется для обработки сразу нескольких командных сообщений с различными идентификаторами. Для этого предназначена макрокоманда ON_COMMAND_RANGE.

Она назначает один метод-обработчик для обработки нескольких командных сообщений, интервалы которых лежат в интервале от id1 до id2:

ON_COMMAND_RANGE(id1,id2,memberFxn)

Макрокоманда ON_UPDATE_COMMAND_UI. Данная макрокоманда обрабатывает сообщения, предназначенные обновления пользовательского

интерфейса, например меню, панелей управления, и позволяет менять их состояние.

Параметр id указывает идентификатор сообщения, а параметр memberFxn - имя метода для его обработки:

ON_UPDATE_COMMAND_UI(id,memberFxn)

Методы, предназначенные для обработки данного класса сообщений, должны быть определены с ключевым словом afx_msg и иметь один параметр - указатель на объект класса CCmdUI. Для удобства имена методов, предназначенных для обновления пользовательского интерфейса, начинаются с OnUpdate:

afx_msg void OnUpdate<имя_обработчика>(CCmdUI* pCmdUI);

В качестве параметра pCmdUI методу передается указатель на объект класса CCmdUI. В нем содержится информация об объекте пользовательского интерфейса, который нужно обновить, - строке меню или кнопке панели управления. Класс CCmdUI также включает методы, позволяющие изменить внешний вид представленного им объекта пользовательского интерфейса.

Сообщения, предназначенные для обновления пользовательского интерфейса, передаются, когда пользователь открывает меню приложения, а также во время цикла ожидания приложения, когда очередь сообщений приложения становится пуста.

При этом посылается несколько сообщений, по одному для каждой строке меню. С помощью макрокоманд ON_UPDATE_COMMAND_UI можно определить методы-обработчики, ответственные за обновление внешнего вида каждой строки меню и соответствующие ей кнопки на панели управления. Эти методы могут изменять состояние строки меню - отображать ее серым цветом, запрещать ее выбор, отображать около нее символ "галочка" и т.д.

Если не определить метод для обновления данной строки меню или кнопки панели управления, то выполняется обработка по умолчанию. Выполняется поиск обработчика соответствующего командного сообщения, и, если он не обнаружен, выбор строки запрещается.

Макрокоманда ON_UPDATE_COMMAND_UI_RANGE. Эта макрокоманда обеспечивает обработку сообщений, предназначенных для обновления пользовательского интерфейса, идентификаторы которых лежат в интервале от id1 до id2. Параметр memberFxn указывает метод, используемый для обработки:

ON_UPDATE_COMMAND_UI_RANGE(id1,id2,memberFxn)

Макрокоманда ON_<name>. Данные макрокоманды предназначены для обработки сообщений от органов управления. Такие сообщения могут передаваться органами управления диалоговой панели. Сообщения от органов управления не имеют обработчиков, используемых по умолчанию. При необходимости их нужно определить самостоятельно.

Все макрокоманды ON_<name> имеют два параметра. В первом параметре id указывается идентификатор органа управления. Сообщения от этого органа управления будут обрабатываться методом memberFxn. Например:

ON_BN_CLICKED(id,memberFxn)

Макрокоманда ON_CONTROL_RANGE. Эта макрокоманда обрабатывает сообщения от органов управления, идентификаторы которых находятся в интервале от id1 до id2. Параметр wNotifyCode содержит код извещения. Метод-обработчик указывается параметром memberFxn:

ON_CONTROL_RANGE(wNotifyCode,id1,id2,memberFxn)

Приложение, обрабатывающее сообщения

Предыдущие два рассматриваемых приложения, фактически никак не могли взаимодействовать с пользователем. Они не имели ни меню, ни панели управления. И, самое главное, они не содержали обработчиков сообщений.

Рассмотрим теперь приложение, которое имеет меню и содержит обработчики сообщений, передаваемых приложению, когда пользователь открывает меню и выбирает из него строки. Пусть меню приложения состоит из одного пункта Test. Можно выбрать одну из следующих команд - Beep или Exit.

Файл ресурсов, в который включается описание меню, можно построить либо непосредственным созданием нового файла ресурсов, либо при помощи средств AppWizard. В любом случае при создании меню нужно определить название меню или строки меню. Каждый элемент меню должен иметь уникальный идентификатор, однозначно его определяющий:

```
//Файл resource.h
    #define IDR_MENU          101
    #define ID_TEST_BEEP   40001
    #define ID_TEST_EXIT   40002
//Файл resource.rc
```

```
#include "resource.h"
IDR_MENU MENU DISCARDABLE
BEGIN
    POPUP "Test"
    BEGIN
        MENUITEM "Beep",      ID_TEST_BEEP
        MENUITEM SEPARATOR
        MENUITEM "Exit",      ID_TEST_EXIT
    END
END
```

Файлы, в которых находятся определение классов приложения и главного окна, представлены ниже:

```
//Файл menu.h
    #include <afxwin.h>
    class CMenuApp: public CWinApp
    {
    public:
        virtual BOOL InitInstance();
    };
//Файл menu.cpp
    #include <afxwin.h>
    #include "menu.h"
    #include "menum.h"
    BOOL CMenuApp::InitInstance()
    {
        m_pMainWnd= new CMainWindow();
        m_pMainWnd->ShowWindow(m_nCmdShow);
        m_pMainWnd->UpdateWindow();
        return TRUE;
    }
    CMenuApp theApp;
//Файл menum.h
    #include <afxwin.h>
    class CMainWindow : public CFrameWnd
    {
    public:
        CMainWindow();
        afx_msg void TestBeep();
        afx_msg void TestExit();
        // макрокоманда необходима, так как класс обрабатывает сообщения
        DECLARE_MESSAGE_MAP()
    };
//Файл menum.cpp
    #include <afxwin.h>
```

```
#include "menum.h"
#include "resource.h"
// Таблица сообщений класса
BEGIN_MESSAGE_MAP(CMainWindow,CFrameWnd)
    ON_COMMAND(ID_TEST_BEEP,TestBeep)
    ON_COMMAND(ID_TEST_EXIT,TestExit)
END_MESSAGE_MAP()
CMainWindow::CMainWindow()
{
    Create(NULL,"Hello",WS_OVERLAPPEDWINDOW,rectDefault,
        NULL,MAKEINTRESOURCE(IDR_MENU));
}
void CMainWindow::TestBeep()// Метод TestBeep - обрабатывает команду меню
{
    MessageBeep(0);
}
void CMainWindow::TestExit()// Метод TestExit - обрабатывает команду меню
{
    DestroyWindow();
}
```

Чтобы объекты класса могли обрабатывать сообщения, в определении класса необходимо поместить макрокоманду DECLARE_MESSAGE_MAP. По принятым соглашениям эта макрокоманда должна записываться в секцию public.

Кроме этого, необходимо также определить таблицу сообщений. Таблица начинается макрокомандой BEGIN_MESSAGE_MAP и заканчивается макрокомандой END_MESSAGE_MAP. Между этими макрокомандами расположены строки таблицы сообщений, определяющие сообщения, подлежащие обработке данным классом, и методы, которые выполняют такую обработку.

Приложение может содержать несколько классов, обладающих собственными таблицами сообщений. Чтобы однозначно определить класс, к которому относится таблица сообщений, имя этого класса записывается в первый параметр макрокоманды BEGIN_MESSAGE_MAP.

Приложение menu обрабатывает только две команды от меню приложения. Для обработки этих команд используют методы, представленные в определении класса CMainFrame.

Приложению может поступать гораздо больше сообщений и команд, чем указано в таблице сообщений класса CMainFrame. Необработанные сообщения передаются для обработки базовому классу - классу CFrameWnd. Класс, который будет обрабатывать сообщения, не указанные в таблице сообщений, указывается во втором параметре макрокоманды BEGIN_MESSAGE_MAP.

Замечание. Если класс приложения тоже обрабатывает сообщения (т.е. имеет таблицу сообщений), и некоторые из сообщений обрабатываются как окном, так и приложением, то нужно понять, какова очередность обработки сообщений тем или иным объектом. Те команды, которые не имеют обработчика в таблице сообщений класса окна, передаются для обработки в класс приложения. Если же команда может быть обработана и в классе окна, и в классе приложения, она обрабатывается только один раз в классе окна. Обработчик класса приложения в этом случае не вызывается.

Некоторые методы класса CWnd

Создание и уничтожение Windows-окон

```
CWnd();
```

Создает объект класса CWnd, обеспечивающий доступ к Windows-окну. При этом само Windows-окно не создается. Далее можно либо создать новое Windows-окно и закрепить его за данным оконным объектом, либо закрепить уже имеющееся Windows-окно. Первое достигается методами CreateEx и Create, второе - методом Attach.

```
virtual BOOL DestroyWindow();
```

Уничтожает Windows-окно, закрепленное за объектом класса CWnd. Если окно уничтожено успешно, возвращается ненулевое значение, в противном случае - 0. После выполнения этого метода оконный объект уже не имеет закрепленного за ним Windows-окна. Сам оконный объект при этом не уничтожается. Метод DestroyWindow посылает соответствующие сообщения, чтобы уничтожить окно и связанные с ним ресурсы. Оно также уничтожает дочерние окна и, если требуется, информирует родительское окно.

Методы инициализации

Методы создания Windows-окон обсуждаться не будут, так как при работе со многими классами окна (к которым относятся и элементы управления) создаются каркасом прилржения. Рассмотрим только методы для прикрепления и открепления Windows-окон.

```
BOOL Attach(HWND hWndNew);
```

Закрепляет Windows-окно, заданное определителем hWndNew за оконным объектом. При успешном выполнении возвращает ненулевое значение, в противном случае - 0.

HWND Detach();

Открепляет Windows-окно, закрепленное за данным оконным объектом, и возвращает определитель открепленного окна.

Методы управления состоянием окна

Перечислим некоторые методы управления состоянием окна.

BOOL IsWindowEnabled() const;

Если окно в выключенном состоянии, возвращает нулевое значение, в противном случае - 0.

BOOL EnableWindow(BOOL bEnable=TRUE);

Если значение параметра TRUE, окно переводится в включенное состояние, FALSE - в выключенное. Метод возвращает ненулевое значение, если в момент вызова окно находилось в выключенном состоянии. Если окно было во включенном состоянии или произошла ошибка, возвращается 0.

CWnd* SetActiveWindow();

Переводит окно в активное состояние. Возвращает указатель на оконный объект, обеспечивающий доступ к окну, активному в момент вызова этого метода (указатель может быть временным и не должен запоминаться для дальнейшего использования).

static CWnd* PASCAL GetActiveWindow();

Возвращает указатель на оконный объект, обеспечивающий доступ к окну, активному в момент вызова этого метода. Если в момент вызова активных окон нет, возвращается NULL. Этот указатель может быть временным и не должен запоминаться для дальнейшего использования.

CWnd* SetCapture();

Переводит окно в состояние захвата мыши. Возвращает указатель на оконный объект, обеспечивающий доступ к окну, которым мышь была захвачена в момент вызова этого метода. Если в момент вызова мышь не захвачена, возвращает NULL. Этот указатель может быть временным и не должен запоминаться для дальнейшего использования. Чтобы освободить

мышь, используется API-функция ReleaseCapture (параметров не имеет). При успешном ее выполнении возвращается TRUE, иначе - FALSE.

static CWnd* PASCAL GetCapture();

Возвращает указатель, задающий окно, захватившее мышь. Если такого окна нет, возвращает NULL.

BOOL ModifyStyle(DWORD dwRemove, DWORD dwAdd, UINT nFlags=0);

Изменяет стиль окна. Параметр dwRemove задает набор элементов стиля, которые должны быть изъяты из стиля окна. Параметр dwAdd - набор элементов стиля, которые должны быть добавлены к стилю окна. Возвращает ненулевое значение, если стиль был успешно изменен, в противном случае - 0.

Если параметр nFlags не равен 0, то после изменения стиля вызывается API-функция SetWindowPos, которая перерисовывает окно, используя набор флагов, полученный комбинацией значения:

- SWP_NOSIZE - сохранять текущий размер;
- SWP_NOMOVE - сохранять текущую позицию;
- SWP_NOZORDER - сохранять текущий Z-порядок;
- SWP_NOACTIVE - не делать окно активным.

Методы управления размером и положением окна

void MoveWindow(int x, int y, int nWidth, int nHeight, BOOL bRepaint=TRUE);

void MoveWindow(LPCRECT lpRect, BOOL bRepaint=TRUE);

Эти методы изменяют положение и размеры окна. Положение левого верхнего угла окна задантся координатами x,y, а размеры шириной nWidth и высотой nHeight. Параметр bRepaint определяет, будет ли инициироваться перерисовка. Если он равен TRUE, окну будет послано сообщение WM_PAINT, в противном случае сообщение не посылается, и перерисовка не производится. Эти действия применяются как к клиентской, так и неклиентской области окна, а также к частям родительского окна, открывшимся при перемещении.

Новое положение и размеры окна можно задать и с помощью структуры типа RECT или объекта класса CRect, передав в качестве параметра ссылку на структуру или объект класса.

Для окна, у которого нет родителя, координаты указываются относительно левого верхнего угла экрана, а для имеющего такового - относительно верхнего левого угла родительского окна.

BOOL SetWindowPos(const CWnd* pWndInsertAfter, int x, int y, int cx, int cy, UINT nFgags);

Изменяет положение, размеры и место окна в Z-упорядочении (порядке изображения окон в слоях изображения). Параметры x, y, cx, cy задают новое положение левой стороны, новое положение верхней стороны, новую длину и новую высоту соответственно.

Параметр pWndInsertAfter определяет окно, за которым нужно поместить исходное окно в Z-упорядочении. Этот параметр может быть либо указателем на объект класса CWnd, либо одним из следующих значений:

- wndBottom поместить окно в конец Z-упорядочения, т.е. позади всех окон на экране;
- wndTop поместить окно в начало Z-упорядочения, т.е. впереди всех окон на экране;
- wndTopMost поместить окно на ближайшее место, делающее окно неперекрытым. Окно перемещается на неперекрытое место, даже если оно неактивно;
- wndNoTopMost поместить окно на ближайшее место позади всех неперекрытых окон. Окно, перекрытое в момент вызова этого метода, не перемещается.

Параметр nFlags задает режим изменения размера и положения окна и может быть следующей комбинацией флагов:

- SWP_DRAWFRAME изображать вокруг окна рамку (определенную при его создании);
- SWP_HIDEWINDOW скрыть окно;
- SWP_NOACTIVE не делать окно активным. Если этот флаг не установлен, окно делается активным и помещается впереди, либо на место в Z-упорядочении, определяемое параметром pWndInsertAfter;
- SWP_NOMOVE сохранить текущее положение окна, проигнорировав параметры x и y;
- SWP_NOREDRAW не перерисовывать измененное окно. Если этот флаг установлен, то после выполнения функции окно с новыми установками на экране не появится, а старое изображение окна не будет стерто с родительского окна;
- SWP_NOSIZE сохранить текущий размер окна, проигнорировав параметры cx и cy;
- SWP_NOZORDER сохранить текущее Z-упорядочение, проигнорировав параметр pWndInsertAfter;
- SWP_SHOWWINDOW показать окно (сделать окно видимым и перерисовать).

Если окно не является дочерним, координаты указываются относительно левого верхнего угла экрана, иначе координаты указываются относительно верхнего левого угла клиентской области родительского окна. Приложение не может активизировать неактивное окно, не поместив его в начало Z-упорядочения. Приложение не может изменить место активного окна в Z-упорядочении произвольным образом. Перекрытые окна могут быть родительскими по отношению к неперекрытым, но не наоборот.

void GetWindowRect(LPRECT lpRect) const;

Копирует параметры прямоугольника, ограничивающего окно, в структуру типа RECT или объект класса CRect, заданные параметром lpRect. Этот прямоугольник включает все - и клиентскую и системную часть окна. Параметры даются относительно левого верхнего угла экрана. При вызове метода значением фактического параметра может быть либо ссылка на (не константную) структуру типа RECT либо объект класса CRect.

void GetClientRect(LPRECT lpRect) const);

Копирует параметры прямоугольника, ограничивающего клиентскую часть окна, в структуру типа RECT или объект класса CRect, определенные параметром lpRect. Параметры даются относительно левого верхнего угла клиентской области окна, поэтому левая и верхняя составляющие будут равны нулю, а правая и нижняя - ширине и длине клиентской области. При вызове метода параметр lpRect может быть либо указателем на структуру типа RECT, либо переменной класса CRect.

Методы взаимодействия Windows-окон

BOOL UpdateData(BOOL bSaveAndValidate=TRUE);

Выполняет обмен данными между объектом класса, производного от CWnd, и частным случаем Windows-окна - диалоговым окном. Если параметр равен FALSE, данные будут пересылаться от объекта и обновлять содержимое элементов управления диалогового окна. В противном случае данные будут считываться из элементов управления диалогового окна и обновлять переменные оконного объекта. При выполнении этого метода происходит вызов виртуального метода DoDataExchange класса CWnd. Эту функция нужно переопределить, чтобы она выполнила весь необходимый обмен. Как правило, в качестве объектов, обменивающихся данными с диалоговым окном, выступают объекты пользовательских классов, производных от CDialog

или CFormView. Для них имеется возможность создания переменных, связанных с элементом управления. В этом случае работу по созданию переопределенной функции DoDataExchange выполняет ClassWizard.

Методы управления текстом окна

```
void SetWindowText(LPCTSTR lpszString);
```

Устанавливает текст заголовка окна. Если окно - элемент управления, устанавливает текст в этом элементе. При вызове функции параметр должен быть либо указателем на строку символов, оканчивающуюся нулевым символом, либо переменной типа CString.

```
int GetWindowText(LPSTR lpszStringBuf, int nMaxCount) const;

void GetWindowText(CString& rString) const;
```

Копирует текст из заголовка окна. Если окно - элемент управления, копирует текст из этого элемента. При вызове первого варианта функции параметр lpszStringBuf должен быть указателем на буфер, в который будет скопирован текст, а параметр nMaxCount - выражением, задающим размер буфера (максимальное число символов, которое разрешается скопировать в буфер). Функция возвращает число скопированных символов, не включающее нуль-символ. При вызове второго варианта параметр rString должен быть переменной типа CString.

```
int GetWindowTextLength() const;
```

Возвращает длину текста заголовка окна, а если окно является элементом управления - длину текста этого элемента. Длина не учитывает нуль-символ.

```
CFont* GetFont() const;
```

Получает текущий шрифт данного окна.

```
void SetFont(CFont* pFont, BOOL bRedraw=TRUE);
```

Устанавливает текущий шрифт окна. Параметр pFont должен задавать новое значение для текущего шрифта. Если параметр bRedraw равен TRUE, окно после установки нового шрифта перерисовывается.

Некоторые методы класса CButton

UINT GetState() const;

Возвращает описание набора текущих состояний кнопки. Чтобы выделить из этого описания значения конкретных типов состояния, можно использовать маски:

- 0x0003 - выделяет собственное состояние кнопки. Применимо только к флажку или переключателю. Если результат побитового умножения дает 0, значит кнопка находится в невыбранном состоянии, 1 - в выбранном, 2 - в неопределенном.
- 0x0004 - выделяет состояние первого типа. Ненулевой вариант означает, что кнопка "нажата", нулевой - кнопка свободна.
- 0x0008 - выделяет положение фокуса. Ненулевой вариант - кнопка в фокусе клавиатуры.

int GetCheck() const;

Возвращает собственное состояние флажка или переключателя. Возвращаемое значение может принимать одно из значений: 0 - кнопка не выбрана; 1 - кнопка выбрана; 2 - кнопка в неопределенном состоянии. Если кнопка не является ни переключателем, ни флажком, возвращается 0.

void SetCheck(int nCheck);

Устанавливает собственное состояние флажка или переключателя. Значения задаются из набора: 0 - невыбранное; 1 - выбранное; 2 - неопределенное. Значение 2 применимо только к флажку со свойством 3State.

UINT GetButtonStyle() const;

Возвращает стиль кнопки.

void SetButtonStyle(UINT nStyle, BOOL bRedraw=TRUE);

Устанавливает стиль кнопки. Если параметр bRedraw равен TRUE, кнопка перерисовывается.

HICON GetIcon() const;

Возвращает дескриптор пиктограммы, сопоставленной кнопке. Если у кнопки нет сопоставленной пиктограммы, возвращает NULL.

HICON SetIcon(HICON hIcon);

Сопоставляет кнопке пиктограмму. Значением параметра при вызове должен быть дескриптор пиктограммы.

Пиктограмма автоматически помешается на поверхность кнопки и сдвигается в ее центр. Если поверхность кнопки меньше пиктограммы, она обрезается со всех сторон до размеров кнопки. Положение пиктограммы может быть выровнено и не по центру. Для этого нужно, чтобы кнопка имела одно из следующих свойств: BS_LEFT, BS_RIGHT, BS_CENTER, BS_TOP, BS_BOTTOM, BS_VCENTER

Данный метод устанавливает для кнопки только одну пиктограмму, которая будет наравне с текстом присутствовать при любом ее состоянии. Не надо путать ее с растровым изображением у растровой кнопки.

HBITMAP GetBitmap() const;

Возвращает дескриптор растрового изображения, сопоставленного кнопке. Если такового не существует, то возвращается NULL.

HBITMAP SetBitmap(HBITMAP hBitmap);

Сопоставляет кнопке растровое изображение. Значением параметра должен быть дескриптор растрового изображения. Правила размещения растрового изображения такие же, как и у значка.

HCURSOR GetCursor();

Возвращает дескриптор курсора, сопоставленного кнопке методом SetCursor. Если у кнопки нет сопоставленного курсора, то возвращается NULL.

HCURSOR SetCursor(HCURSOR hCursot);

Сопоставляет кнопке курсор, изображение которого будет помещено на поверхность кнопки аналогично значку и растровому изображению.

Некоторые методы класса CEdit

Окна редактирования могут работать в режимах однострочного и многострочного редакторов. Приведем сначала методы, общие для обоих режимов, а затем методы для многострочного редактора.

Общие методы

```
DWORD GetSel() const;

void GetSel(int& nStartChar, int& nEndChar) const;
```

Получает первую и последнюю позиции выделенного текста. Для значения типа DWORD младшее слово содержит позицию первого, старшее - последнего символа.

```
void SetSel(DWORD dwSelection, BOOL bNoScroll=FALSE);

void SetSel(int nStartChar, int nEndChar, BOOL bNoScroll=FALSE);
```

Устанавливает новое выделение текста, задавая первый и последний выделенный символ. Значение FALSE параметра bNoScroll должно отключать перемещение курсора в область видимости.

```
void ReplaceSel(LPCTSTR lpszNewText);
```

Заменяет выделенный текст на строку, передаваемую в параметре lpszNewText.

```
void Clear();
```

Удаляет выделенный текст.

```
void Copy();
```

Копирует выделенный текст в буфер.

```
void Cut();
```

Переносит (копирует и удаляет) выделенный текст в буфер обмена.

```
void Paste();
```

Вставляет текст из буфера обмена, начиная с позиции, в которой находится курсор.

BOOL Undo();

Отмена последней операции, выполненной редактором. Если редактор однострочный, возвращается всегда неотрицательное значение, иначе неотрицательное значение возвращается лишь в случае успешной замены.

BOOL CanUndo() const;

Определяет, можно ли отменить последнюю операцию редактора.

void EmptyUndoBuffer();

Сбрасывает флаг undo, сигнализирующий о возможности отмены последней операции редактора, и тем самым делает невозможным отмену. Этот флаг сбрасывается автоматически при выполнении методов SetWindowText и SetHandle.

BOOL GetModify() const;

Возвращает неотрицательное значение, если содержимое окна редактирования не модифицировалось. Информация о модификации поддерживается в специальном флаге, обнуляемом при создании окна редактирования и при вызове метода:

void SetModify(BOOL bModified=TRUE);

Устанавливает или сбрасывает флаг модификации (см. предыдущий метод). Флаг сбрасывается при вызове метода с параметром FALSE и устанавливается при модификации содержимого окна редактирования или при вызове SetModify с параметром TRUE.

BOOL SetReadOnly(BOOL bReadOnly=TRUE);

Устанавливает режим просмотра (bReadOnly=TRUE) или редактирования (bReadOnly=FALSE).

TCHAR GetPasswordChar() const;

Возвращает символ, который при выводе пароля будет появляться на экране вместо символов, набираемых пользователем. Если такой символ не определен, возвращается 0. Устанавливается этот символ методом (по умолчанию используется "*"):

```
void SetPasswordChar(TCHAR ch);
```

```
void LimitText(int nChars=0);
```

Устанавливает максимальную длину в байтах текста, который может ввести пользователь. Если значение параметра равно 0, длина текста устанавливается равной UINT_MAX.

Методы работы с многострочным редактором

```
void LineScroll(int nLines, int nChars=0);
```

Прокручивает текст в области редактирования. Параметр nLimes задает число строк для вертикальной прокрутки. Окно редактирования не прокручивает текст дальше последней строки. При положительном значении параметра область редактирования сдвигается вдоль текста к последней строке, при отрицательной - к первой.

Параметр nChars задает число символов для горизонтальной прокрутки. Окно редактирования прокручивает текст вправо, даже если строки закончились. В этом случае в области редактирования появляются пробелы. При положительном значении параметра область редактирования сдвигается вдоль к концу строки, при отрицательном - к началу.

```
int GetFirstVisibleLine() const;
```

Возвращает номер первой видимой строки.

```
int GetLineCount() const;
```

Возвращает число строк текста, находящегося в буфере редактирования. Если текст не вводился, возвращает 1.

```
int GetLine(int nIndex, LPTSTR lpszBuffer) const;
```

```
int GetLine(int nIndex, LPTSTR lpszBuffer, int nMaxLength) const;
```

Копирует строку с номером, равным значению параметра nIndex, в буфер, заданный параметром lpszBuffer. Первое слово в буфере должно задавать его размер. При вызове второго варианта метода значение параметра nMaxLength копируется в первое слово буфера.

Метод возвращает число в действительности скопированных байтов. Если номер строки больше или равен числу строк в буфере окна редактирования, возвращает 0. Текст копируется без каких-либо изменений, нуль-символ не добавляется.

```
int LineIndex(int nLine=-1) const;
```

Возвращает номер первого символа в строке. Неотрицательное значение параметра принимается в качестве номера строки. Значение -1 задает текущую строку. Если номер строки больше или равен числу строк в буфере окна редактирования (строки нумеруются с 0), возвращается 0.

Некоторые методы класса CListBox

```
void ResetContent();
```

Очищает содержимое списка, делая его пустым.

```
int AddString( LPCSTR lpszItem);
```

Добавляет строку lpszItem в список и сортирует его, если при создании включено свойство Sort. В противном случае элемент добавляется в конец списка.

```
int DeleteString( UINT nIndex);
```

Удаляет из списка элемент с индексом nIndex. Индексация элементов начинается с 0.

```
int GetCurSel() const;
```

Получает индекс элемента, выбранного пользователем.

```
int SetCurSel( int nSelect);
```

Отмечает элемент с индексом nSelect как выбранный элемент списка. Если значение параметра равно -1, список не будет содержать отмеченных элементов.

```
int GetText( int nIndex, LPSTR lpszBuffer) const;
```

```
void  GetText( int nIndex, CString& rString) const;
```

Копирует элемент с индексом nIndex в буфер.

```
int SetTopIndex( int nIndex);
```

Организует прокрутку списка в окне так, чтобы элемент с индексом nIndex был видимым.

```
int FindString( int nStartAfter, LPCSTR lpszItem) const;
```

Организует поиск в списке и возвращает в качестве результата индекс элемента списка, префикс которого совпадает со строкой lpszItem. Результат не зависит от регистра, в котором набирались символы сравниваемых строк. Параметр nStartAfter задает начало поиска, но поиск идет по всему списку. Он начинается от элемента, следующего за nStartAfter, до конца списка и затем продолжается от начала списка до элемента с индексом nStartAfter. В качестве результата выдается первый найденный элемент, удовлетворяющий условиям поиска. Если такого нет, результат получает значение LB_ERR.

```
int FindStringExact( int nIndexStart, LPCSTR lpszFind) const;
```

Этот метод отличается от предыдущего тем, что теперь не префикс элемента должен совпадать со строкой lpszFind, а сам элемент. Поиск по-прежнему не чувствителен к регистру, в котором набираются символы.

Диагностика объектов.

Диагностический сервис помогает при отладке приложений. Для диагностики объектов в MFC существуют макросы, позволяющие печатать отладочную информации во время выполнения приложения, отслеживать распределение памяти, содержимое дампа (dump) объекта.

Макросы

ASSERT (booleanExpression) - прерывает выполнение программы, если выражение booleanExpression=FALSE, печатая при этом сообщение об ошибке.

ASSERT_KINDOF (className, pObject) - проверяет, является ли pObject объектом класса className. className — имя класса, производного от класса CObject. Этот макрос работает только, если в области объявления класса используется один из следующих макросов: DECLARE_ DYNAMIC или DECLARE_SERIAL.

ASSERT_VALID (pObject) - проверяет внутреннее состояние объекта при помощи сравнения pObject с NULL, после чего вызывает его функцию-член AssertValid. При возникновении ошибки при одной из двух проверок, выводится сообщение об ошибке.

TRACE (exp) - выводит на экран форматированную строку exp, аналогично функции printf(), например: TRACE ("Hello %s. Year %d", "world", 2000) выведет на экран строку "Hello world. Year 2000."

TRACE0, ..., TRACE3 - аналогичны TRACE. Отличие состоит в том что эти макросы позволяют вывести форматированную строку с числом аргументов от 1 до 3 соответственно.

Вышеперечисленные макросы работают только в отладочной (debug) версии приложения.

VERIFY (booleanExpression) — макрос, аналогичный макросу ASSERT, только для рабочей (release) версии приложения.

Графические объекты

Библиотека MFC обеспечивает разработчиков всеми необходимыми классами, которые инкапсулируют соответствующие графические объекты Windows. Кроме того, библиотека имеет в своем составе дополнительные классы, значительно облегчающие решение ряда задач(классы CPoint, CSize, CRect, CRectTracker) или унифицирующие работу с графическими объектами как таковыми(CGdiObject).

Классы графических объектов

Класс CGdiObject

Базовый класс для всех классов, обеспечивающий интерфейс с графическими объектами Windows.

Класс CPen

Инкапсулирует объект Windows «карандаш», который может быть выбран в контекст устройства и использоваться для определения типа и цвета линий или графических фигур.

Класс CBrush

Инкапсулирует объект Windows «кисть», который может быть выбран в контекст устройства и использоваться для определения типа и цвета заливки внутренних областей замкнутых фигур.

Класс CFont

Инкапсулирует объект Windows «шрифт», который может быть выбран в контекст устройства и использоваться при операциях вывода текстовой информации.

Создать объект класса CPen или CBrush можно двумя способами:

1)Конструктор используется как для создания собственно объекта, так и для его инициализации.

Пример

```
CPen pen(PS_DOT,1,RGB(255,0,0));
CBrush brush(HS_CROSS,RGB(0,255,0));
```

2) Конструктор используется только для создания объекта, а для его инициализации дополнительно вызывается функции (эта функции имеет префикс Create)

Пример

```
CPen pen;
pen.CreatePen(PS_SO:ID,2,RGB(200,150,50));
CBrush brush;
brush.CreateSolidBrush(RGB(0,0,255));
```

Чтобы настроить параметры рисования с помощью соответствующего графического объекта, необходимо выполнить следующие действия

1)Создать графический объект.

2)Заменить в контексте устройства текущий графический объект вновь созданным, сохранив при этом указатель на «старый» объект.

3)Закончив операции рисования, восстановить «старый» графический объект в контексте устройства при помощи сохраненного указателя.

4)Обеспечить удаление созданного объекта при выходе из области видимости, что происходит автоматически для объектов, созданных в стеке приложения.

Установка графических объектов

Установка объектов рисования выполняется функцией SelectObject()

```
CPen* SelectObject( CPen* pPen );
CBrush* SelectObject( CBrush* pBrush );
```

```
virtual CFont* SelectObject( CFont* pFont );
```

Кроме графических объектов, созданных в приложении, можно использовать и предопределенные системные. Для установки системных графических объектов используется функция SelectStockObject( )

```
virtual CGdiObject* SelectStockObject( int nIndex );
```

Параметр nIndex задает тип создаваемого объекта.

Отображение графических фигур

Для отображения графических фигур можно использовать следующие функции-члены класса CDC:

Прямоугольник(квадрат)

```
BOOL Rectangle( int x1, int y1, int x2, int y2 );
BOOL Rectangle( LPCRECT lpRect );
```

Эллипс(окружность)

```
BOOL Ellipse( int x1, int y1, int x2, int y2 );
BOOL Ellipse( LPCRECT lpRect );
```

Сегмент

```
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 );
```

Сектор

```
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 );
```

Фигуры рисуются установленным «карандашом» CPen и закрашиваются кистью CBrush.

Вывод текста

Для вывода текста можно использовать функции-члены класса CDC-DrawText() и TextOut() , инкапсулирующие соответствующие функции Windows

```
int DrawText( LPCTSTR lpszString, int nCount, LPRECT lpRect, UINT nFormat );
int DrawText( const CString& str, LPRECT lpRect, UINT nFormat );
virtual BOOL TextOut( int x, int y, LPCTSTR lpszString, int nCount );
BOOL TextOut( int x, int y, const CString& str );
```

Цвет символов и фона текста устанавливаются следующими функциями

```
CDC::virtual COLORREF SetTextColor( COLORREF crColor );
CDC::virtual COLORREF SetBkColor( COLORREF crColor );
```

Для создания шрифта для вывода текста необходимо создать объект класса TFont, проинициализировать его и установить в контексте устройства.

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

Пример

```
CFont font;
LOGFONT lf;
memset(&lf, 0, sizeof(LOGFONT));
lf.lfHeight = 12;
strcpy(lf.lfFaceName, "Arial");
VERIFY(font.CreateFontIndirect(&lf));
CClientDC dc(this);
CFont* def_font = dc.SelectObject(&font);
dc.TextOut(5, 5, "Hello", 5);
dc.SelectObject(def_font);
font.DeleteObject();
```

CFont::BOOL 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);

Пример

```
CFont font;
VERIFY(font.CreateFont(
 12,                  // nHeight
 0,                   // nWidth
```

```
 0,                    // nEscapement
 0,                    // nOrientation
 FW_NORMAL,                // nWeight
 FALSE,                // bItalic
 FALSE,                // bUnderline
 0,                    // cStrikeOut
 ANSI_CHARSET,             // nCharSet
 OUT_DEFAULT_PRECIS,       // nOutPrecision
 CLIP_DEFAULT_PRECIS,      // nClipPrecision
 DEFAULT_QUALITY,          // nQuality
 DEFAULT_PITCH | FF_SWISS,  // nPitchAndFamily
 "Arial"));                // lpszFacename
CClientDC dc(this);
CFont* def_font = dc.SelectObject(&font);
dc.TextOut(5, 5, "Hello", 5);
dc.SelectObject(def_font);
font.DeleteObject();
BOOL CreatePointFont( int nPointSize, LPCTSTR lpszFaceName, CDC* pDC = NULL );
```

Пример

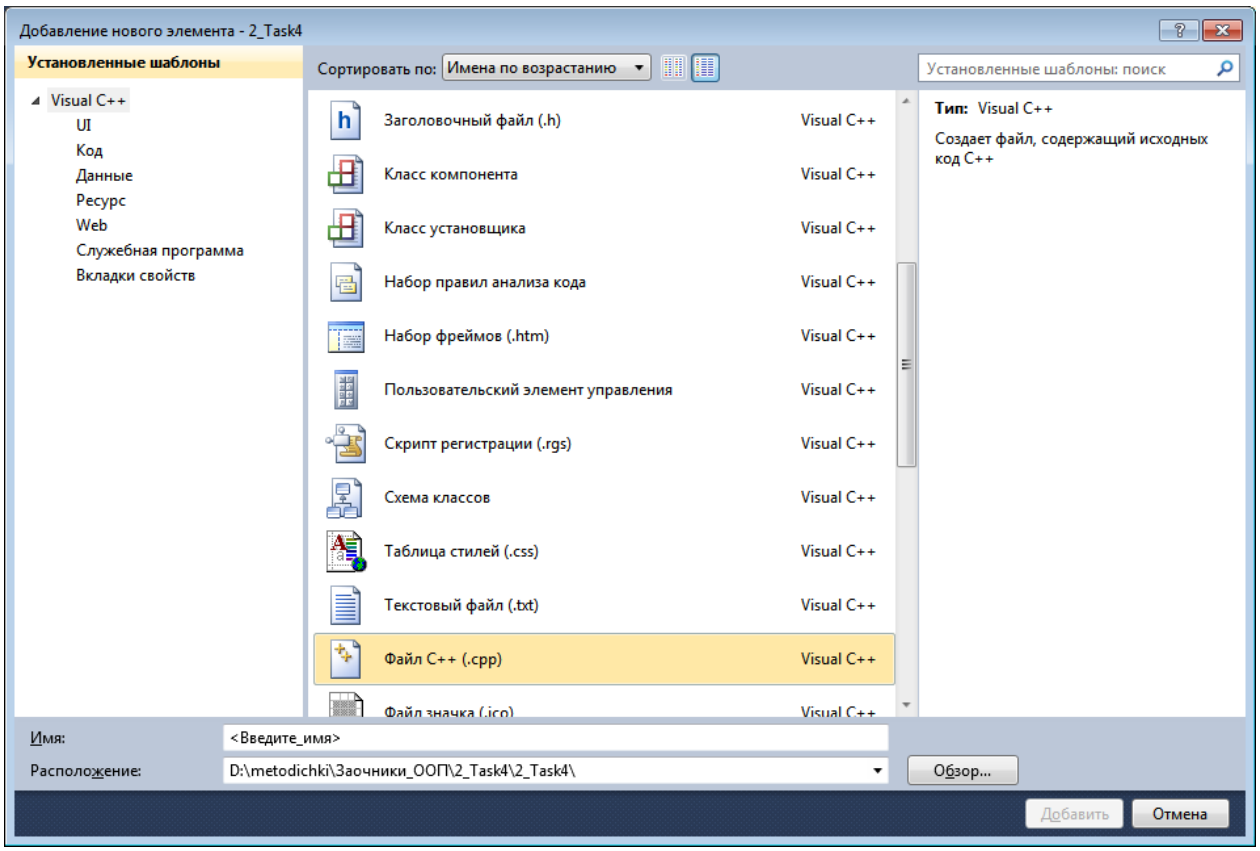
```
CClientDC dc(this);
CFont font;
VERIFY(font.CreatePointFont(120, "Arial", &dc));
CFont* def_font = dc.SelectObject(&font);
dc.TextOut(5, 5, "Hello", 5);
dc.SelectObject(def_font);
font.DeleteObject();
```

Методические указания

1.Загружаем Visual Studio и выбираем Файл->Создать ->Проект.

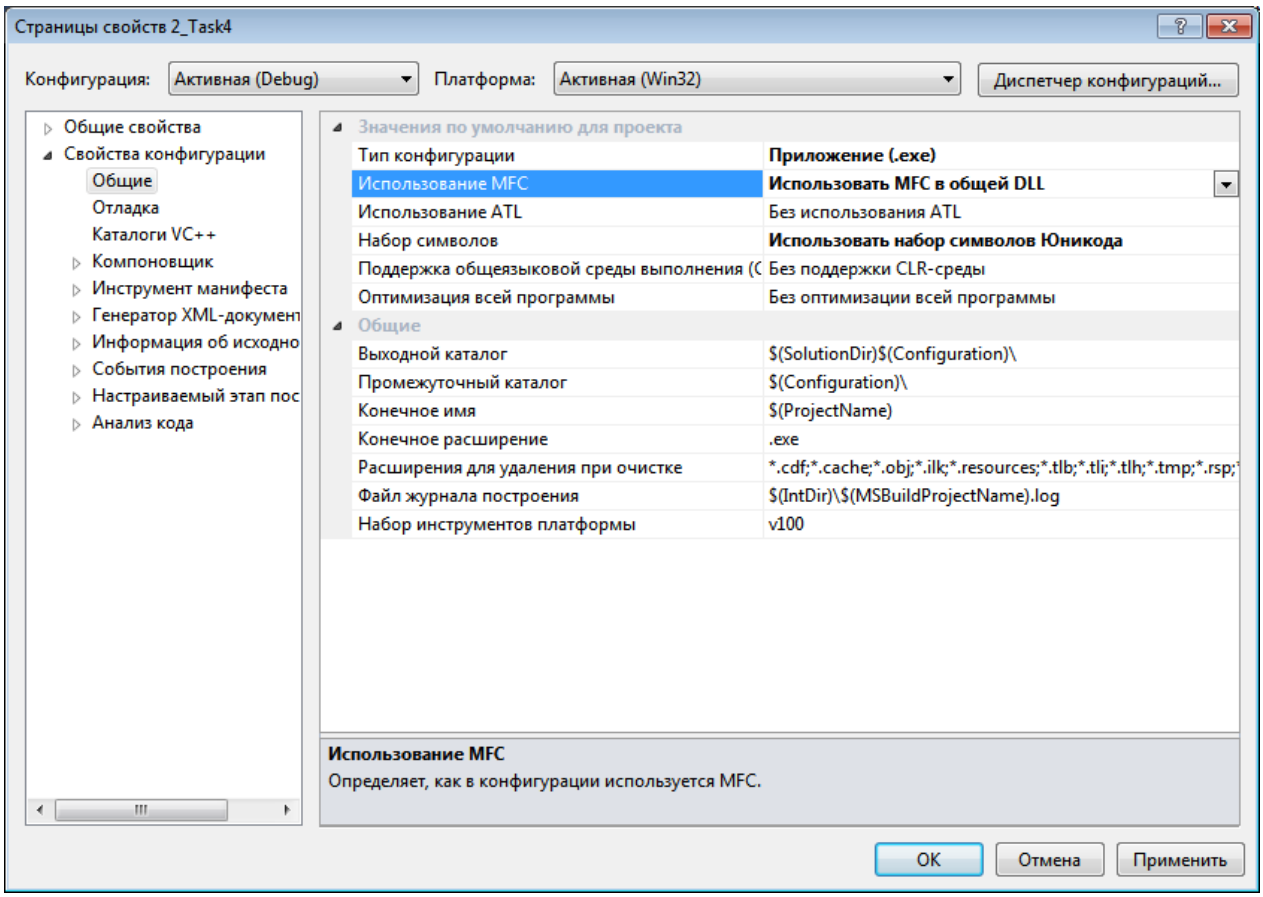
2.Выбираем тип проекта Win32. Этот тип - приложение под Windows, но без использования MFC. MFC же мы подсоединим позже. Выберите пункт пустой проект.

3.Добавим в проект файлы.



4. Укажем явно, что мы хотим использовать MFC. Для этого заходим в Свойства

Выбираем **Использовать MFC в общей DLL**. Теперь наш проект будет использовать MFC.

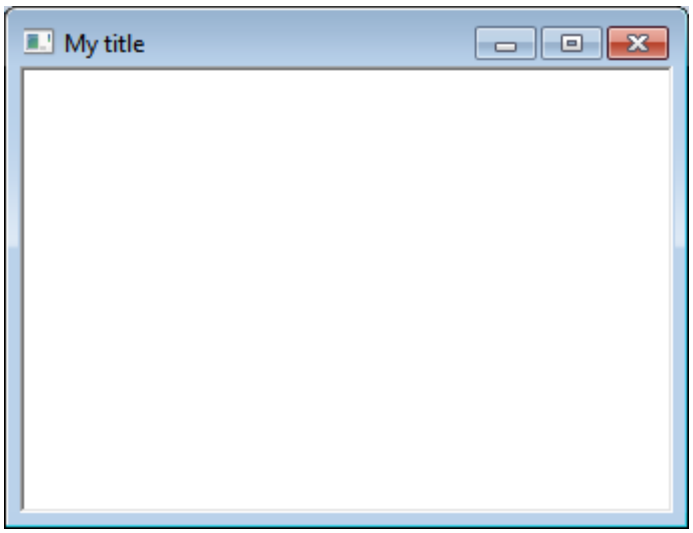


5.Вставьте в файл вашего приложения следующий код:

```
#include <afxwin.h>
class CMyMainWnd : public CFrameWnd{
public:
	CMyMainWnd(){ // конструктор
		Create(NULL,L"My title");
	}
};
class CMyApp : public CWinApp{
public:
	CMyApp(){}; // конструктор
	virtual BOOL InitInstance(){
		m_pMainWnd=new CMyMainWnd();
		m_pMainWnd->ShowWindow(SW_SHOW);
		return TRUE;
	}
};
CMyApp theApp;
```

6.Запустите программу на выполнение

Должно появиться окно с заголовком "My title".



7.Разберем код.

Мы создали два класса - CMyMainWnd и CMyApp. Первый из них задаёт главное окно нашего приложения. Второй - само приложение. В конце нашего кода в строке

CMyApp theApp;

мы создаём экземпляр нашего приложения.

В классе главного окна ничего кроме контруктора нет. В конструкторе мы вызываем метод Create, который наш класс окна наследует от родительского класса.

В классе CMyApp переопределяется функция InitInstance родительского класса. В ней в строке

m_pMainWnd= new CMyMainWnd();

динамически создается новый экземпляр нашего главного окна. В следующей строке

m_pMainWnd->ShowWindow(SW_SHOW);

наше созданное окно показывается на экране.

8.Сделаем так, чтобы программа обращала внимание на наши действия. Например, чтобы при щелчке мышкой появлялся MessageBox.

Для этого в наш класс окна вставьте следующий строчки(они выделены):

```
class CMyMainWnd : public CFrameWnd{
public:
        CMyMainWnd(){ // конструктор
                Create(NULL,L"My title");
        }

        afx_msg void OnLButtonDown(UINT, CPoint);
  DECLARE_MESSAGE_MAP()
```

```
};
```

После определения класса запишите

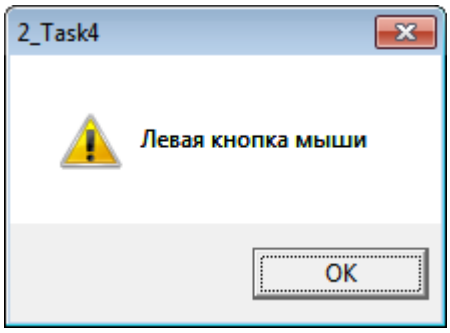
```
BEGIN_MESSAGE_MAP(CMyMainWnd, CFrameWnd)
ON_WM_LBUTTONDOWN()
END_MESSAGE_MAP()
```

И, наконец, в конце файла добавьте строки

```
CMyApp theApp;


void CMyMainWnd::OnLButtonDown(UINT, CPoint){
   AfxMessageBox(L"Левая кнопка мыши");
}
```

9.Откомпилируйте и выполните приложение. При нажатии левой кнопки мыши в окне появляется MessageBox с надписью "Левая кнопка мыши".



10.Разберем код.

Для того, чтобы наш класс обращал внимание на наши действия, мы должны сделать следующие действия.

Первое. Мы должны вставить в конец нашего класса макрос DECLARE_MESSAGE_MAP(). Это достаточно сделать один раз. Этот макрос в классе и означает, что этот класс будет реагировать на некоторые сообщения.

Второе. Мы должны где-то после класса добавить два макроса BEGIN_MESSAGE_MAP(...,...) и END_MESSAGE_MAP(). Это тоже достаточно сделать только один раз. Это так называемая карта сообщений. В первый макрос первым параметром вы должны вставить имя вашего класса, вторым - имя родительского класса. Первый параметр показывает, для какого класса мы пишем нашу карту сообщений, а второй - кто должен обрабатывать сообщение, которое наш класс обработать не может.

Третье. В классе пишем метод для обработки конкретного сообщения. Для стандартных сообщений имена методов стандартны. Например, для

сообщения WM_ONLBUTTONDOWN имя метода-обработчика OnLButtonDown. Перед названием метода не забудем написать afx_msg. В нашем примере это

afx_msg void OnLButtonDown(UINT, CPoint);

Четвёртое. В карту сообщений пишем макрос для нашего сообщения. В нашем примере это строка ON_WM_LBUTTONDOWN(). Его имя - это ON_ плюс имя сообщения.

```
BEGIN_MESSAGE_MAP(CMyMainWnd, CFrameWnd)
ON_WM_LBUTTONDOWN()
END_MESSAGE_MAP()
```

Пятое. Записываем код метода. Здесь мы для примера написали

```
void CMyMainWnd::OnLButtonDown(UINT, CPoint){
    AfxMessageBox("Левая кнопка мыши");
}
```

Функции с префиксом Afx определены в MFC как глобальные

11. Нарисуем что-нибудь в окне.

Когда окну надо что-либо перерисовать, оно получает сообщение WM_PAINT. Для рисования нам надо написать обработчик для этого события.

Вносим объявление функции-обработчика события WM_PAINT в класс окна:

```
class CMyMainWnd : public CFrameWnd{
public:
    CMyMainWnd(){ // конструктор
        Create(NULL,L"My title");
    }

    afx_msg void OnLButtonDown(UINT, CPoint);
    afx_msg void OnPaint();
  DECLARE_MESSAGE_MAP()

};
```

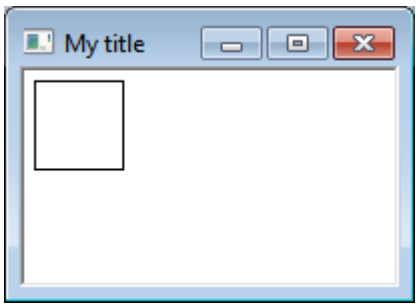
Затем добавляем макрос в карту сообщений:

```
BEGIN_MESSAGE_MAP(CMyMainWnd, CFrameWnd)
ON_WM_LBUTTONDOWN()
ON_WM_PAINT()
END_MESSAGE_MAP()
```

И, наконец, пишем реализацию нашей функции:

```
void CMyMainWnd::OnPaint(){
   CPaintDC* pDC=new CPaintDC(this);
   pDC->Rectangle(5,5,50,50);
}
```

Откомпилируем и выполним программу. В левом углу должен появится квадратик.



12.Добавим в программу обработку таймера, т.е. события WM_TIMER.

Для этого, во-первых, создадим таймер. Для этого в конструктор класса CMyMainWnd добавим следующий код:

```
CMyMainWnd(){ // конструктор
                Create(NULL,L"My title");
                   SetTimer(1, 1000, NULL);
        }
```

Во-вторых, напишем обработчик события WM_TIMER.

Для этого вы должны сделать три действия - добавить соответствующий метод в класс, написать его реализацию и добавить соответствующий макрос в карту сообщений.

Добавим метод в класс:

```
      ...
   afx_msg void OnPaint();
   afx_msg void OnTimer(UINT);
   DECLARE_MESSAGE_MAP()
```

Напишем реализацию этого метода:

```
void CMyMainWnd::OnTimer(UINT){
   MessageBeep(-1);
      }
```

В-третьих, добавим макрос:

```
...
BEGIN_MESSAGE_MAP(CMyMainWnd, CFrameWnd)
ON_WM_LBUTTONDOWN()
ON_WM_PAINT()
ON_WM_TIMER()
END_MESSAGE_MAP()
```

13.Выполним программу. Каждую секунду должен издаваться звук beep.

14.Так как таймер системный ресурс, в конце программы его надо удалить. Для этого вносим класс код для деструктора:

```
~CMyMainWnd(){
  KillTimer(1);
}
```

15.Заставим программу работать одновременно с двумя таймерами. Добавляем в программу ещё один таймер и сразу пишем код в деструкторе класса для уничтожения нового таймера:

```
        CMyMainWnd(){ // конструктор
                Create(NULL,L"My title");
                  SetTimer(1, 1000, NULL);
                        SetTimer(2, 3000, NULL);
        }
        ~CMyMainWnd(){
  KillTimer(1);
        KillTimer(2);
}
```

Идентификатор нового таймера 2 , и он тикает раз в три секунды.

16.Отдельный обработчик для второго таймера писать не надо, а надо изменить обработчик для таймера следующим образом:

```
void CMyMainWnd::OnTimer(UINT nIDEvent){
  if(nIDEvent==1)
    MessageBeep(-1);
  else
    SetWindowText(L"Title");
}
```

У метода OnTimer есть параметр типа UINT. Это есть идентификатор таймера, для которого мы обрабатываем сообщение WM_TIMER. Если сообщение поступило от первого таймера, то издаём сигнал, а если от второго, то меняем заголовок окна на "Title".

17.Выполним программу. Звук раздаётся раз в секунду, и через три секунды заголовок окна меняется

18. Измените интервал у таймера, т. е. с делайте, чтобы сначала он тикал с одной частотой, а затем с другой. Частота должна меняться по щелчку правой кнопки мыши.

Принцип здесь простой - сначала надо убить старый таймер, а затем создать новый с таким же идентификатором.

Для этого напишете обработчик события WM_RBUTTONDOWN, откомпилируйте и выполните программу.

Пример

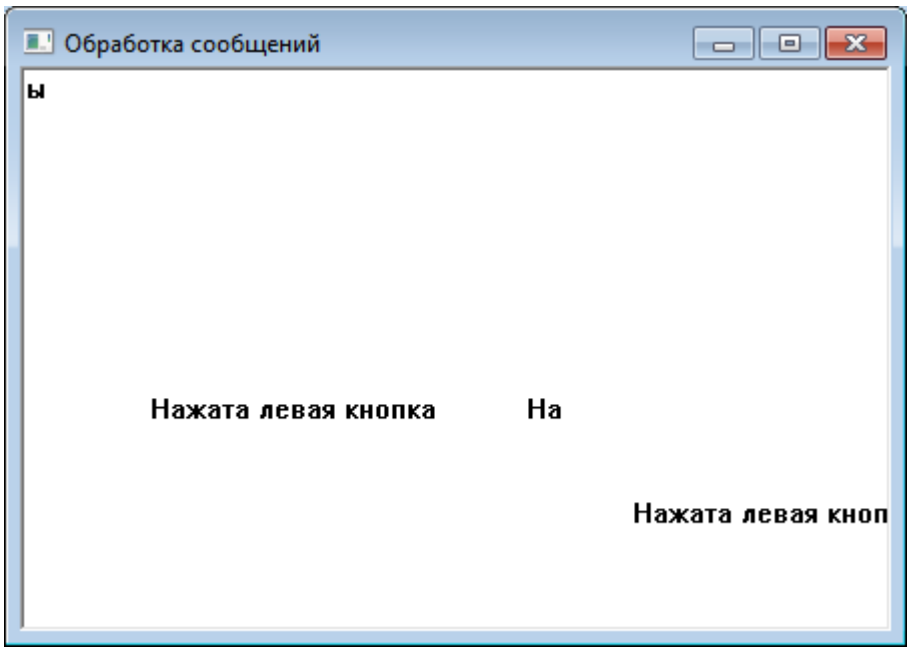
```
#include <afxwin.h>
#include <cstring>
class CMainWin: public CFrameWnd
 {
public:
            CMainWin();
            afx_msg void OnChar(UINT ch, UINT, UINT);
            afx_msg void OnPaint();
            afx_msg void OnLButtonDown(UINT flags, CPoint Loc);
            afx_msg void OnRButtonDown(UINT flags, CPoint Loc);
            char str[50];
            int nMouseX, nMouseY, nOldMouseX, nOldMouseY;
            char pszMouseStr[50];
DECLARE_MESSAGE_MAP()
};
class CApp: public CWinApp
{           public:
BOOL InitInstance();
};
BOOL CApp::InitInstance()
            {
            m_pMainWnd = new CMainWin;
            m_pMainWnd->ShowWindow(SW_RESTORE);
            m_pMainWnd->UpdateWindow();
            return TRUE;        }
CMainWin::CMainWin()
            {
            this->Create(0, "Обработка сообщений");
            strcpy(str, "");
            strcpy(pszMouseStr, "");
            nMouseX = nMouseY = nOldMouseX = nOldMouseY = 0;
            }

BEGIN_MESSAGE_MAP
```

```
        (CMainWin /* класс */, CFrameWnd /* базовый класс */)
        ON_WM_CHAR()
        ON_WM_PAINT()
        ON_WM_LBUTTONDOWN()
        ON_WM_RBUTTONDOWN()
END_MESSAGE_MAP()

afx_msg void CMainWin::OnChar(UINT ch, UINT, UINT)
        {       sprintf(str, "%c", ch);
                this->InvalidateRect(0);
        }
afx_msg void CMainWin::OnPaint()
        {       CPaintDC dc(this);
                dc.TextOut(nOldMouseX, nOldMouseY,      " ", 30);
        dc.TextOut(nMouseX, nMouseY, pszMouseStr);
                dc.TextOut(1,       1, " ");
                dc.TextOut(1, 1, str);
        }
afx_msg void CMainWin::OnLButtonDown
                                        (UINT, CPoint loc)
        {
        nOldMouseX = nMouseX;
        nOldMouseY = nMouseY;
        strcpy(pszMouseStr, "Нажата левая кнопка");
        nMouseX = loc.x; nMouseY = loc.y;
        this->InvalidateRect(0);
        }
afx_msg void CMainWin::OnRButtonDown
                                        (UINT, CPoint loc)
        {
        nOldMouseX = nMouseX;
        nOldMouseY = nMouseY;
        strcpy(pszMouseStr, "Нажата правая кнопка");
        nMouseX = loc.x; nMouseY = loc.y;
        this->InvalidateRect(0);
}       CApp App;
```

Результат выполнения


Обработка сообщений
ы
Нажата левая кнопка На
Нажата левая кноп