

Министерство образования и науки Российской Федерации
Федеральное агентство по образованию
Ярославский государственный университет им. П.Г. Демидова

В.В. Васильчиков

Программирование в Visual C++ с использованием библиотеки MFC

Учебное пособие

*Рекомендовано
Научно-методическим советом университета
для студентов специальности Математическое обеспечение
и администрирование информационных систем*

Ярославль 2006

УДК 004.4
ББК 3 973.2-018я73
В 19

*Рекомендовано
Редакционно-издательским советом университета
в качестве учебного издания. План 2006 года*

Рецензенты:

кандидат физико-математических наук С.И. Щукин;
кафедра теории и методики обучения информатике
ЯГПУ им. К.Д. Ушинского

В 19 **Васильчиков, В.В.** Программирование в Visual C++ с использованием библиотеки MFC : учебное пособие / В.В. Васильчиков ; Яросл. гос. ун-т. – Ярославль : ЯрГУ, 2006. – 236 с.
ISBN 5-8397-0463-6

Рассмотрены основные моменты разработки Windows-приложений в среде Visual C++ с использованием библиотеки MFC.

Рекомендуется студентам, обучающимся по специальности 010503 Математическое обеспечение и администрирование информационных систем (дисциплина "Программирование в среде Windows" (курс по выбору), блок ОПД), очной формы обучения.

Библиогр.: 4 назв.

УДК 004.4
ББК 3 973.2-018я73

ISBN 5-8397-0463-6

© Ярославский
государственный
университет, 2006
© В.В. Васильчиков, 2006

Введение

Система программирования Visual C++ относится к числу наиболее распространенных и популярных средств разработки программного обеспечения. Это высокоуровневая и удобная система, предлагающая широкий набор разнообразных инструментов проектирования.

Данное учебное пособие написано на основе лекционного курса по использованию Visual C++ и библиотеки MFC для создания Windows-приложений, читавшегося автором для студентов факультета ИВТ ЯрГУ, обучающихся по специальности "Математическое обеспечение и администрирование информационных систем".

Автор исходит из предположения, что студенты, приступающие к изучению данного курса, знакомы с языком программирования C++. Предполагается также, что в процессе обучения студенты будут выполнять все предлагаемые им в данном пособии учебные задания. Часть заданий представляет собой развитие или модификацию ранее разработанного приложения. В этом случае в качестве стартовой точки используется предыдущая версия проекта. В тексте задания та часть программного кода, которая должна быть добавлена или модифицирована, выделена полужирным шрифтом. Основная часть заданий взята из книги Майкла Янга [1].

Для удобства использования все исходные коды (точнее проекты Visual C++) доступны в локальной сети факультета. Проекты структурированы по темам учебного курса. Предполагаемый результат находится в папке Solution. Если задание предполагает модификацию разработанной ранее версии приложения, то она находится в папке Starter.

Тема 1. Установка программного обеспечения

Если на вашем компьютере не установлена система программирования Microsoft Visual C++ 6, то ее следует установить. Существует три версии Visual C++: Standard, Professional и Enterprise Edition. Мы предполагаем, что устанавливается последняя из перечисленных редакций. Ниже описывается последовательность шагов для выполнения установки и перечень минимально необходимых компонентов.

1.1. Установка Microsoft Visual C++ 6

Вставьте установочный компакт-диск в дисковод. Если у вас отключена функция Autorun, то вручную запустите программу Run.exe из корневого каталога. Если на вашем диске имеются оригинальная и русифицированная версия системы, рекомендуется устанавливать англоязычную версию.

Если в начале установки вам будет сделано напоминание о том, что для корректной установки следует в установочном меню выбрать только Visual C++ (а не Visual Basic, Visual FoxPro и т.п.), то так и следует поступить. Это напоминание не относится к дополнительным установочным компонентам, таким как ActiveX, Data Access и т.д.

Далее вам будет предложено выбрать директорию для распаковки временных файлов и запустить самораспаковывающийся архивный файл. Сделайте это.

На следующем шаге (в окне "Microsoft Virtual Machine for Java") установите соответствующий флажок и нажмите кнопку Next.

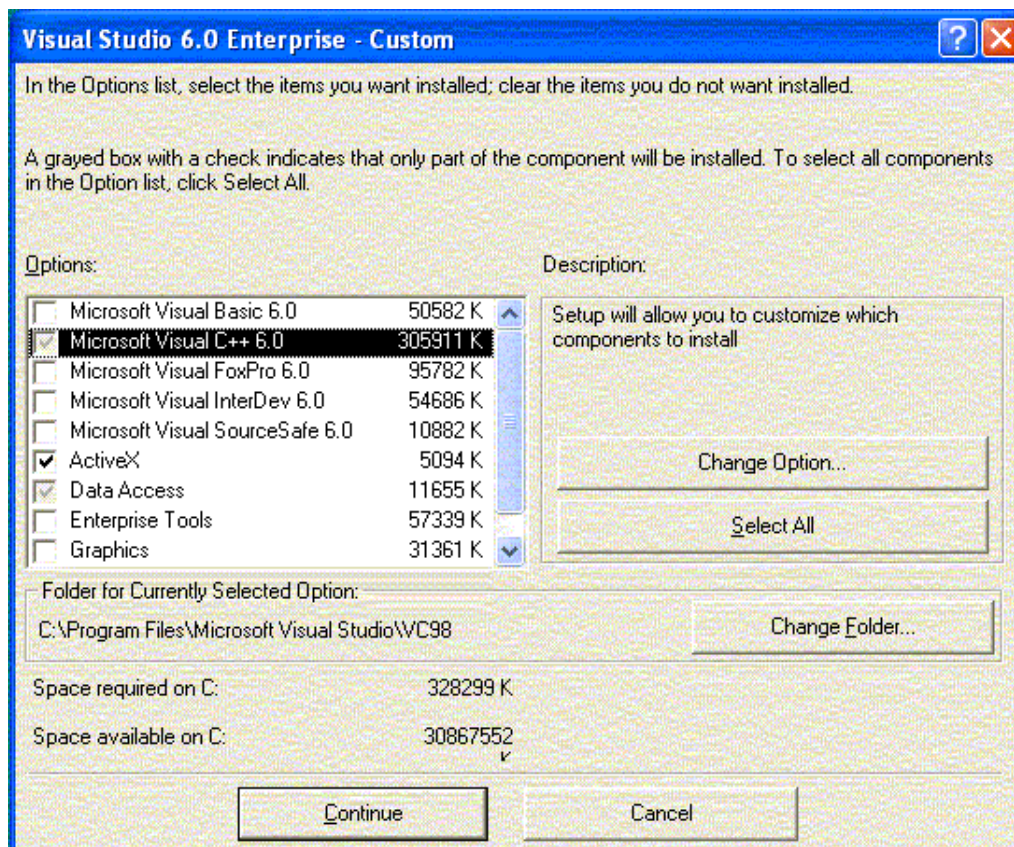
В окне выбора опций установки выберите вариант Custom и нажмите кнопку Next.

В окне "Choose Common Install Folder" выберите подходящую папку и нажмите кнопку Next.

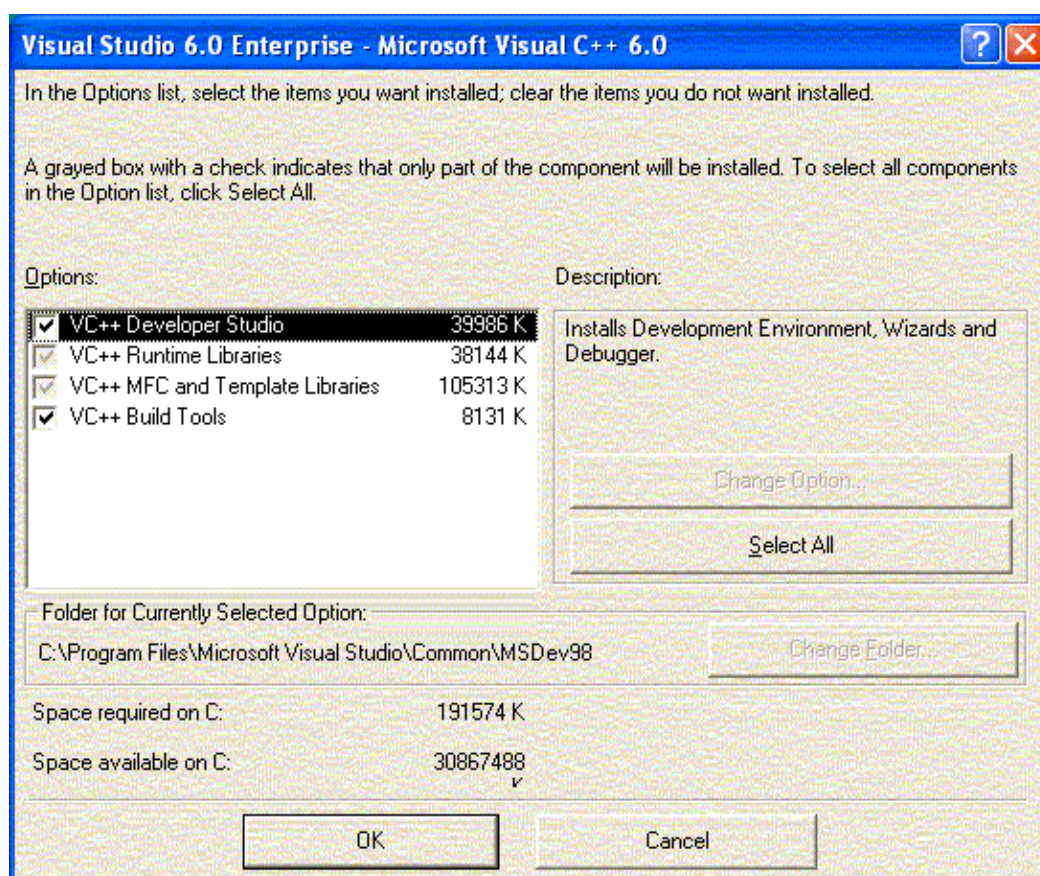
В следующем окне вам будет предложено выбрать устанавливаемые компоненты, в частности, вы можете дополнительно отметить для установки пункты:

- ActiveX (дополнительные элементы ActiveX);
- Data Access (средства доступа к данным)
- Enterprise Tools (инструментальные средства версии Enterprise Edition – если устанавливается эта версия);
- Graphics (графические средства);
- Tools (инструментальные средства).

Отметьте необходимые инструменты. Минимально необходимый набор отмечен на следующем рисунке. Нажмите кнопку Continue.



На следующем шаге отметьте устанавливаемые подкомпоненты Visual C++ в соответствии со следующим рисунком и нажмите кнопку ОК.



Краткое описание выбранных компонентов:

- *VC++ Developer Studio*. Это основа программного продукта Visual C++. Он предоставляет разработчику полный набор инструментов программирования: менеджера проекта, текстовый редактор, мастеров для генерации исходных кодов программы, отладчик и т.д.
- *VC++ Runtime Libraries*. Это библиотеки периода выполнения, которые содержат стандартные функции, используемые в программах на C и C++.
- *VC++ MFC and Template Libraries*. MFC – это обширная библиотека классов, содержащая средства для организации графического интерфейса и множество других компонентов высокого уровня. Template Libraries – библиотеки шаблонов. Можно, например, установить библиотеку шаблонов элементов ActiveX – ATL (Active Template Library).
- *VC++ Build Tools*. В эту группу входят оптимизирующий компилятор VC++, компилятор ресурсов для построения меню и диалоговых окон, а также другие инструментальные средства.

Дождитесь завершения установки системы. После этого вам будет выдано окно, в котором можно зарегистрировать необходимые переменные окружения для запуска Visual C++ из командной строки. Если вы собираетесь пользоваться этой возможностью, то установите соответствующий флажок.

Если же вместо голой командной строки вы предпочитаете пользоваться оболочками типа Far manager, то обратите внимание на упомянутый в окне файл VCVARS32.BAT. Там содержатся команды для необходимых настроек переменных окружения. Его можно скопировать в свою папку, переименовать, например, в Far_VC.bat, добавить в конце вызов Far manager и использовать в дальнейшем для вызова этой оболочки.

Наконец, последнее окно, выводимое программой установки Visual C++, позволяет после ее завершения начать установку справочной системы MSDN. Для этого следует установить соответствующий флажок и нажать кнопку Next.

1.2. Установка справочной системы Visual C++ 6

Первое окно установочной программы MSDN предлагает задать путь для установки, а также выбрать один из трех возможных ее вариантов. Произведем выборочную (custom) установку.

Второе (и последнее) окно программы позволяет выбрать устанавливаемые разделы справочной системы. В нашем случае достаточно ограничиться отметкой трех разделов:

- Full Text Search Index;
- VC++ Documentation;
- Microsoft Platform SDK Documentation.

Теперь достаточно нажать кнопку Continue и дождаться завершения установки.

Тема 2. Создание программ в среде Developer Studio

Ниже приводятся сведения, минимально необходимые для того, чтобы начать разработку собственных программ в среде Developer Studio. Предполагается, что в процессе работы студенты овладеют существенно более широким спектром возможностей данной среды разработки.

Создание проекта

Опишем последовательность действий для создания простейшего проекта, например, заготовки для разработки консольного приложения. Для этого требуется:

- Запустить среду Developer Studio
- Выбрать в меню File команду New
- На вкладке Project выбрать тип проекта (в нашем случае – Win32 Console Application), его размещение, дать проекту имя
- Выбрать вид генерируемого кода (в нашем случае – пустой проект)

Создание и редактирование исходного файла программы

- Выбрать в меню File команду New ...
- На вкладке Files выбрать тип файла (в нашем случае – C++ Source File), его размещение, дать файлу имя (в нашем случае – Hello.cpp).
- Ввести код программы, воспользовавшись текстовым редактором. При редактировании файлов удобно пользоваться вкладками FileView и ClassView, а также интерактивной справочной системой. Пример текста:

```
// Hello.cpp: The C++ source code for the HelloUser program.
#include <iostream.h>
char Name [16];
void main ()
{
    cout << "Enter your name: ";
    cin.getline (Name, sizeof (Name));
    cout << "\nVery glad to see you, " << Name << "\n";
    cout << "\nPress Enter to continue...";
    cin.get ();
}
```

Некоторые возможности текстового редактора

Интегрированный в среду Developer Studio текстовый редактор предоставляет разработчику множество удобных возможностей для создания и редактирования текста программы. Он, естественно, поддерживает все привычные возможности работы с буфером обмена Windows. Не перечис-

для всех его возможностей, автору хочется упомянуть некоторые из них, с его точки зрения, наиболее часто употребляемые именно при создании программного кода на C++.

Операция	Комбинация клавиш
Переход на дополняющую скобку	Указатель перед скобкой и Ctrl+]
Сдвинуть строки вправо	Выделить строки и Tab
Сдвинуть строки влево	Выделить строки и Shift+Tab
Открыть диалоговое окно Find	Ctrl+F
Найти следующее вхождение	F3
Найти предыдущее вхождение	Shift+F3
Найти след. вх. выделенного текста	Ctrl+F3
Найти пред. вх. выделенного текста	Ctrl+Shift+F3
Включить/выключить закладку	Ctrl+F2
Перейти к следующей закладке	F2
Перейти к предыдущей закладке	Shift+F2
Снять все закладки	Ctrl+Shift+F2

Отладка программы

Как и в любой другой среде программирования, удобство разработки во многом определяется возможностями предоставляемого отладчика. Для того, чтобы запустить процесс отладки, необходимо проделать следующие действия:

1. Открыть проект, остановить в качестве текущей конфигурации Win32 Debug, произвести сборку проекта.
Для этого можно воспользоваться меню Build, клавишей Build на панели инструментов Build или Build MiniBar, или просто нажав клавишу F7.
2. Произвести отладку, воспользовавшись панелью отладчика (включение/выключение – меню Tools/Customize, вкладка Toolbars: отметить Debug), либо комбинациями клавиш, а также возможностями настройки точек останова (меню Edit/Breakpoints).
Для запуска отладчика можно воспользоваться меню Build/Start Debug, клавишей Go на панели инструментов Build или Build MiniBar, или нажав клавишу F5. Комбинация Ctrl-F5 используется для запуска программы на выполнение без отладчика.

Из дополнительных возможностей отладки хочется обратить внимание студентов на использование макросов TRACE (доступны только при поддержке MFC) и ASSERT. Для их использования необходимо задать некоторые дополнительные параметры сборки проекта. Сначала нужно вы-

звать диалоговое окно Project Settings (нажав, например, Alt-F7), затем на вкладке C/C++ в строке Preprocessor definitions добавить определение _AFXDLL (если его там нет), а в окне Project Options установить опцию /MDd.

В процессе отладки разработчик может отобразить на экране следующие окна:

- QuickWatch – Shift+F9, панель отладчика или правая кнопка мыши
- Watch – View/Debug Windows, панель отладчика, обычно включено
- Variables – View/Debug Windows, панель отладчика, обычно включено
- CallStack – View/Debug Windows, панель отладчика
- Memory – View/Debug Windows, панель отладчика
- Registers – View/Debug Windows, панель отладчика
- Disassembly – View/Debug Windows, панель отладчика

Комбинации клавиш отладчика Developer Studio

Ниже перечислены некоторые наиболее употребляемые в процессе отладки комбинации клавиш.

Операция	Комбинация клавиш
Добавить/удалить точку останова в строке с курсором	F9
Удалить все точки останова	Ctrl+Shift+F9
Начать/возобновить выполнение программы	F5
Повторить выполнение программы с самого начала	Ctrl+Shift+F5
Выполнить следующий оператор (режим step into)	F11
Выполнить следующий оператор (режим step over)	F10
Выполнить до выхода из текущей функции (step into)	Shift+F11
Выполнить до текущей позиции курсора	Ctrl+F10
Перейти к позиции курсора без выполнения операторов	Ctrl+Shift+F10
Открыть диалоговое окно QuickWatch	Shift+F9
Открыть диалоговое окно Breakpoints	Ctrl+B
Конец отладки	Shift+F5

В качестве упражнения для освоения перечисленных средств создания и отладки программ в среде Developer Studio читателям предлагается по-пробовать их на примере разработки какого-либо простого консольного приложения, например, вычисления суммы числового ряда с заданной точностью. Рекомендуется попробовать использование макросов TRACE и ASSERT.

Тема 3. Модель программирования в Windows

Материал данной темы предназначен в основном для тех, кто ранее программировал только в MS DOS. Для краткости ограничимся тем, что перечислим основные отличия модели программирования в Windows, понимать которые необходимо для разработки Windows-приложений.

Обработка сообщений

При программировании в MS DOS на языках C и C++ выполнение программы начиналось с вызова функции `main()`, а та уже в свою очередь вызывала при необходимости другие функции. Для того чтобы узнать, например, нажата ли какая-либо клавиша на клавиатуре, программа обращалась к соответствующим средствам операционной системы (обычно через вызов библиотечных функций).

Главная функция Windows-приложения – `WinMain()`. Ее задача – создание основного окна программы, с которым должен быть связан код, способный обрабатывать сообщения операционной системы (`WM_CREATE`, `WM_CHAR`, `WM_LBUTTONDOWN` и т.п.). Таким образом, не программа вызывает функции операционной системы, а операционная система посылает приложению уведомления о произошедших событиях (нажата клавиша, сдвинулась мышь и т.п.). Сообщение имеет два 32-разрядных параметра для передачи дополнительной информации. За обработку сообщений отвечает *каркас приложений* (application framework).

Интерфейс графического устройства

Приложения MS DOS зачастую осуществляли запись данных непосредственно в видеопамять или порт принтера. Следовательно, существовала необходимость использования подходящих драйверов для каждой модели видеоплаты или принтера, что, естественно, жизни программиста не облегчало.

В Windows существует особый "слой абстракции" – интерфейс графического устройства (Graphic Device Interface, GDI). Программа вызывает GDI-функции, ссылающиеся на структуру данных "*контекст устройства*" (device context), а Windows выдает команды ввода/вывода для соответствующего устройства.

Программирование, основанное на ресурсах

При программировании в MS DOS данные зачастую определялись с помощью инициализирующих констант, либо считывались из отдельных файлов. При работе в Windows вы храните данные в файлах ресурсов, которые предоставляются в нескольких стандартных форматах, например:

- *битовые карты (bitmaps), значки (icons)*
- *определения меню (menu definitions)*
- *описания структуры диалоговых окон*
- *строки*
- *особые форматы, определяемые программистом*

Для редактирования этих данных, как правило, используются специальные инструменты, обеспечивающие режим WYSIWYG.

Динамически подключаемые библиотеки

При работе в MS DOS объектные модули программы связываются статически на стадии компоновки. Windows позволяет осуществлять *динамическое связывание*, т.е. загрузку и подключение к программе во время выполнения *динамически подключаемых библиотек* (Dynamic-Link Library, DLL). Тем самым обеспечиваются следующие преимущества:

- экономия памяти и дискового пространства
- улучшение модульности (DLL компилируются и тестируются раздельно).

Например, все классы каркаса приложений MFC реализованы как набор DLL. Их можно подключать статически либо динамически, а также создавать свои DLL-модули расширения (extension DLLs).

Интерфейс прикладных программ Win32 API

Win32 API – это набор базовых функций, обеспечивающих прямое общение с операционной системой, ускоряющих работу и уменьшающих размер приложения. Использование *каркаса приложений* позволяет ускорить разработку и в значительной степени избавить от проблем перехода от одной версии API к другой.

Процесс разработки Windows-приложений значительно упрощается за счет использования специальных библиотек, представляющих собой высокоуровневую надстройку над Win32 API:

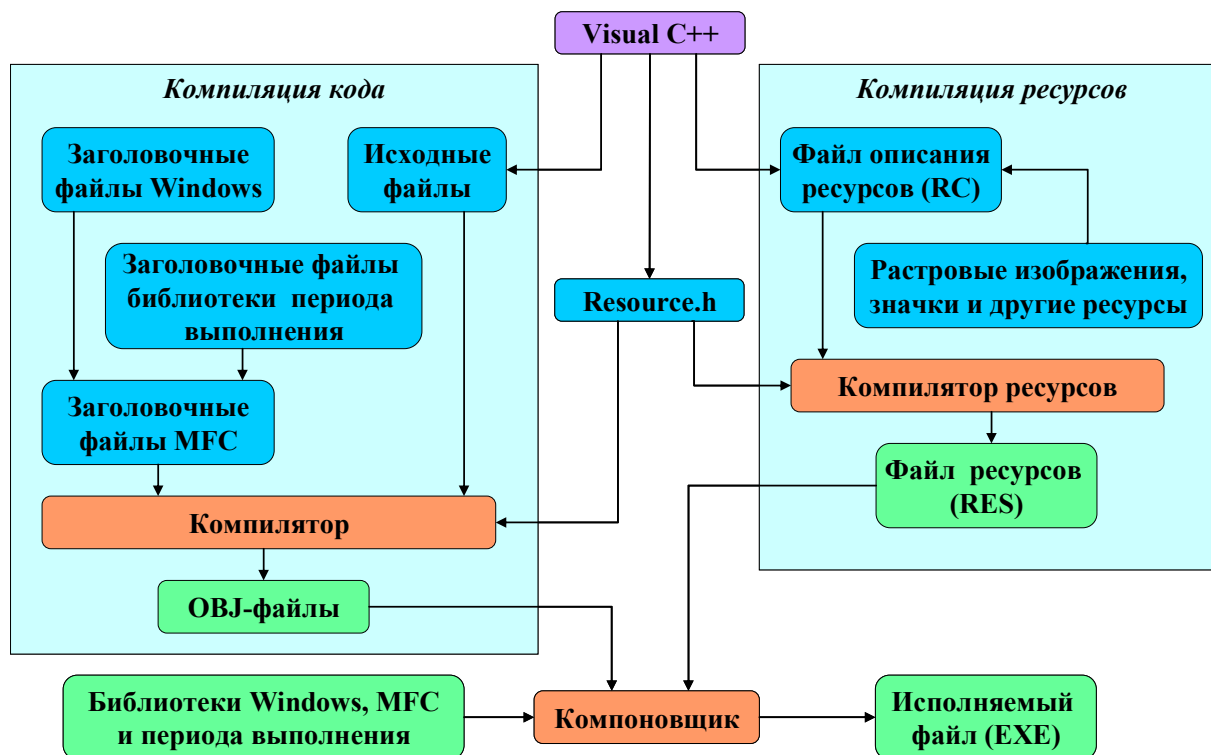
- MFC – Microsoft Foundation Classes
- ATL – Active Template Library (для технологии COM)
- WFC – Windows Foundation Classes (используется для языка Java)

Мы в процессе дальнейшей работы будем использовать средства, предоставляемые библиотекой MFC.

Тема 4. Процесс построения программ в Visual C++

4.1. Создание программы в Visual C++

Следующий рисунок иллюстрирует процесс создания программ в Visual C++ и назначение отдельных файлов проекта.



Проект программы

Проект (project) – набор взаимосвязанных файлов, компиляция и компоновка которых позволяет создать исполняемую Windows-программу или DLL.

Одним из возможных вариантов получения исполняемого файла является использование *сборочного файла проекта* (make file) и *программы управления сборкой проекта* (make program). Среда Visual C++ позволяет создать сборочный файл (с расширением .mak – по умолчанию отсутствует), но, как правило, для управления сборкой проекта используются следующие файлы:

- текстовый файл проекта (project file) – DSP
- текстовый файл рабочего пространства (workspace file) – DSW

Чтобы начать работать с проектом, достаточно открыть соответствующий DSW-файл.

Промежуточные файлы Visual C++

Среда Visual C++ при работе также создает промежуточные файлы нескольких типов. Эти типы и назначение файлов перечислены в следующей таблице.

Расширение файла	Описание
APS	Поддержка обзора ресурсов (ResourceView)
BSC	Информация средства просмотра
CLW	Поддержка ClassWizard
DEP	Файл зависимостей
DSP	Файл проекта
DSW	Файл рабочего пространства
MAK	Внешний сборочный файл
NCB	Поддержка обзора классов (ClassView)
OPT	Содержит конфигурацию рабочего пространства
PLG	Файл журнала сборки

4.2. Компоненты Visual C++

В данном параграфе мы вкратце рассмотрим основные средства разработки программ, входящие в состав интегрированной среды Visual Studio 6.0, а именно:

- Редакторы и средства просмотра ресурсов
- Компилятор C/C++
- Редактор исходного текста
- Компилятор ресурсов
- компоновщик
- Отладчик
- AppWizard
- ClassWizard
- Средства просмотра исходного кода
- Интерактивная справочная система
- Components and Controls Gallery

Редакторы и средства просмотра ресурсов

Открыв вкладку *ResourceView* в окне проектов Visual C++, можно осуществить редактирование ресурсов данного проекта. Разработчику доступны следующие инструменты:

- WYSIWYG – редактор меню
- Графический редактор диалоговых окон. Позволяет также вставлять элементы управления ActiveX. Позволяет обрабатывать EXE- и DLL-файлы
- Дополнительные инструменты для работы со значками, растровыми изображениями и строками

В каждом проекте обычно есть один RC-файл (текстовый), который содержит описания ресурсов меню, диалоговых окон, строк и акселераторов. Он может содержать также директивы #include для сборки ресурсов из других файлов или каталогов (обычно BMP, ICO, строки сообщений). Обычно не рекомендуют модифицировать RC-файл вручную без помощи графического редактора, хотя иногда это позволяет осуществить более точную настройку размеров и положения визуальных компонентов.

Компилятор C/C++

Позволяет обрабатывать исходный код на C и C++. Расширение C соответствует коду на C, а CPP или CXX – коду на C++.

Поддерживает шаблоны, обработку исключений и идентификацию типов во время выполнения (runtime type identification, RTTI).

Имеется *стандартная библиотека шаблонов* (Standard Template Library, STL).

Редактор исходного текста

Поддерживает выделение цветом синтаксических конструкций, автоотступы, сдвиги блоков текста, поиск парных скобок и т.п. Чуть подробнее его возможности мы описывали выше.

Компилятор ресурсов

Назначение: из текстового (RC) файла описания ресурсов создать для компоновщика двоичный (RES) файл.

Компоновщик

Считывает OBJ- и RES-файлы, а также LIB-файлы, содержащие MFC-код, код библиотек периода выполнения и Windows.

Формирует EXE-файл.

Процесс ускоряется за счет компоновки с приращением (incremental link).

AppWizard

AppWizard – генератор кода, создающий рабочую заготовку (но не код целиком) Windows-приложения с теми компонентами, именами классов и исходных файлов, которые вы задаете в диалоге с ним.

Можно создать собственного мастера AppWizard (на основе макросов).

ClassWizard

Реализован, как DLL и доступен через меню View среды разработчика. С его помощью удобно создать класс, тело функции, код, связывающий сообщение Windows с конкретной функцией, и т.п.

Средства просмотра исходного кода

ClassView (вкладка) – позволяет увидеть дерево всех классов проекта вместе с функциями-членами и переменными-членами, а также быстро перейти в соответствующее место исходного кода.

Source Browser (доступен через меню Tools) позволяет также выяснить, где описана и используется функция, переменная, макрос или класс, базовые и производные классы и т.п.

Интерактивная справочная система

Основана на HTML и реализована как отдельное приложение – MSDN Library Viewer.

Вызов: меню Help или F1 (контекстная справка).

Components and Controls Gallery

Позволяет использовать одни и те же программные компоненты в нескольких проектах.

Поддерживает три типа модулей:

- Элементы управления ActiveX. Они регистрируются при установке в реестре Windows. После этого любой может быть выбран и использован в программе.
- Модули исходного кода на C++. Создав класс, его можно добавить в Components Gallery, а потом вставить в новый проект. Так же можно добавлять ресурсы.
- Компоненты VisualC++. Поставляются в виде OGX-файлов. Часть включена в состав VisualC++, часть поставляется сторонними фирмами.

Тема 5. Создание программ с графическим интерфейсом

Можно предложить несколько подходов для создания Windows-приложений с графическим интерфейсом:

- На С или С++ с непосредственным обращением к функциям Win32 API. Это очень трудоемкий подход, причем множество усилий тратится на организацию интерфейса, а не на решение целевой задачи.
- С помощью библиотеки MFC, содержащей большой набор готовых классов и вспомогательный код для стандартных задач программирования (создание окон, обработка сообщений и т.п.). Используется также для быстрого добавления в программы панелей инструментов, многопанельных окон, поддержки OLE, создания элементов ActiveX. Это более высокоуровневый подход, при этом допускается и использование функций Win32 API.
- На С или С++ с использованием MFC и различных мастеров. Среда Developer Studio предлагает использование мастера AppWizard для генерации основы исходных файлов программы. Мастер ClassWizard используется для генерации основной части кода для определения производных классов и обработчиков сообщений, настройки библиотеки MFC, управления диалоговыми окнами и т.д. Основное преимущество такого подхода заключается в избавлении программиста от множества рутинных операций.

Мы в своей работе в рамках данного учебного курса будем использовать третий подход. Далее для освоения основных операций по созданию Windows-приложения в рамках данной схемы студентам предлагается выполнить следующие шаги. Результатом их выполнения станет однодокументное (SDI) приложение, построенное в стиле Document/View.

Генерация исходного кода

Шаг 0: выбор в меню File пункта New.

- указать на вкладке Projects тип (здесь – MFC AppWizard (exe)) и имя (здесь – WinGreet).

Шаг 1:

- выбор типа приложения (здесь – Single document).
- выбор поддержки архитектуры Document/View. В результате будут сгенерированы отдельные классы для хранения (а также чтения и записи на диск) и отображения данных программы.

Шаг 2:

- отключить поддержку баз данных.

Шаг 3:

- выбрать пункт None, чтобы исключить поддержку составных документов
- отключить поддержку автоматизации
- не добавлять поддержку ActiveX

Шаг 4:

- оставить только флажок 3D controls и значение 4 (по умолчанию) для задания количества в списке последних открывавшихся файлов.

Шаг 5:

- выбрать MFC standard – традиционный пользовательский интерфейс. Вариант Windows Explorer – приложение оформляется в виде контейнера, подобного блокноту.
- выбрать генерацию комментариев в исходных файлах.
- выбрать статическую компоновку с MFC.

Шаг 6:

- выбор имен создаваемых классов и файлов для их реализации (здесь – оставить без изменения).

Изменение исходного кода

Открыть файл WinGreetDoc.h (описание класса, ответственного за хранение документа). Добавить в описание класса:

```
class CWinGreetDoc : public CDocument
{
protected:
    char *m_Message;
public:
    char *GetMessage() { return m_Message; }

protected: // create from serialization only
// ...
}
```

1. Открыть файл WinGreetDoc.cpp (реализация класса документа). Добавить в конструкторе класса:

```
CWinGreetDoc::CWinGreetDoc()
{
    // TODO: добавьте код конструктора
    m_Message = "Greetings!";
}
```

2. Открыть файл WinGreetView.cpp (реализация класса представления программы – порождается от класса CView). Добавить код отображения данных:

```

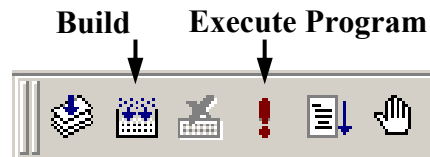
void CWinGreetView::OnDraw(CDC* pDC)
{
    CWinGreetDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);

    // TODO: добавьте код отображения собственных данных
    RECT ClientRect;
    GetClientRect(&ClientRect);
    pDC->DrawText(
        pDoc->GetMessage(), // получить строку
        -1,
        &ClientRect,
        DT_CENTER | DT_VCENTER | DT_SINGLELINE);
}

```

Для компиляции и компоновки приложения можно воспользоваться одним из трех способов:

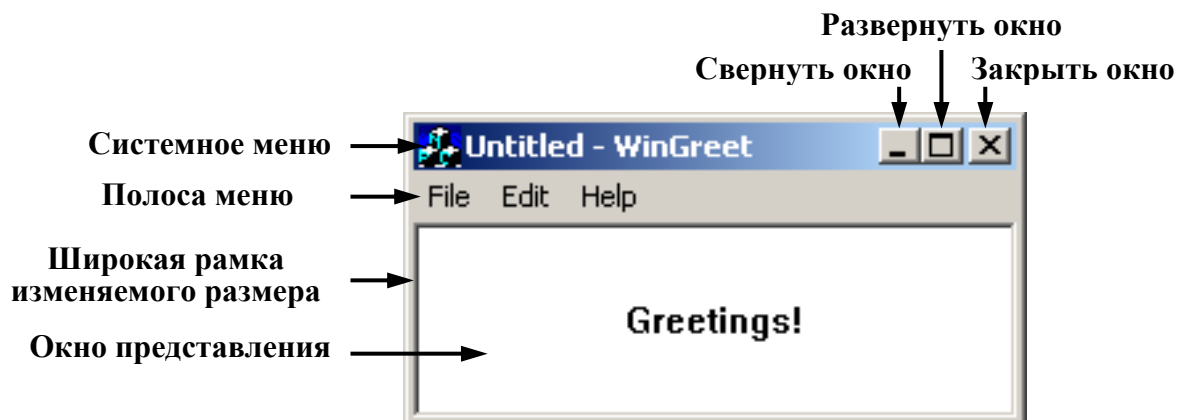
- в меню Build выбрать команду Build WinGreet.exe
- нажать клавишу F7
- нажать кнопку Build на панели инструментов (см. рисунок)



Для запуска приложения (без отладчика) можно также выбрать один из трех вариантов:

- в меню Build выбрать команду Execute WinGreet.exe
- нажать клавиши Ctrl-F5
- нажать кнопку Execute на панели инструментов (см. рисунок).

Внешний вид главного окна запущенного приложения показан на рисунке. Там же отмечены основные элементы данного окна.



Разберемся, какой вариант исходного кода был сгенерирован мастером AppWizard. Был создан код отображения меню, а также реализованы пунк-

ты меню File/Edit и Help/About. Остальные команды или не были реализованы (меню Edit), или были реализованы частично (меню File). В сгенерированном варианте при выборе команд меню File отображаются диалоги открытия и сохранения файлов, но реально чтения и записи не происходит (при сохранении в существующий файл он уничтожается). Таким образом, код чтения и сохранения данных необходимо писать самостоятельно. Реализовано также запоминание до 4 имен файлов, открывавшихся последними.

Классы и файлы программы

При создании SDI-приложения мастер AppWizard создает 4 главных класса, реализованные в отдельных файлах:

- класс документа (Doc). Он отвечает за хранение данных программы, за чтение и запись на диск. Порождается от класса CDocument библиотеки MFC.
- класс представления (View). Порождается от MFC-класса CView. Отвечает за отображение данных программы (на экране, принтере или другом устройстве) и за обработку информации, вводимой пользователем. Он управляет окном представления (view window).
- класс приложения (App). Порождается от MFC-класса CWinApp. Управляет программой в целом. Решает общие задачи, например, инициализацию программы и ее заключительную очистку. Каждая MFC-программа должна создать в точности один экземпляр класса, порожденного от CWinApp.
- класс главного окна. По умолчанию называется CMainFrame и порождается от класса CFrameWnd библиотеки MFC. Управляет главным окном программы, которое является обрамляющим окном и содержит рамку окна, строку заголовка, строку меню и системное меню. Оно содержит кнопки Minimize, Maximize, Close, может содержать также другие элементы пользовательского интерфейса, например, панель инструментов, строку состояния и т.п. Окно представления (в клиентской части главного окна) является его дочерним окном и не содержит никаких видимых элементов, кроме текста и графики, отображающихся явно.

Этапы выполнения программы

Чтобы лучше разобраться, как работает созданное нами приложение, разберем последовательность основных событий при запуске программы WinGreet, а именно:

- Вызов конструктора класса CWinApp
- Получение управления функцией WinMain
- Вызов функции InitInstance функцией WinMain
- Обработка сообщений функцией WinMain
- Выход из функции WinMain и завершение программы

Вызов конструктора класса CWinApp

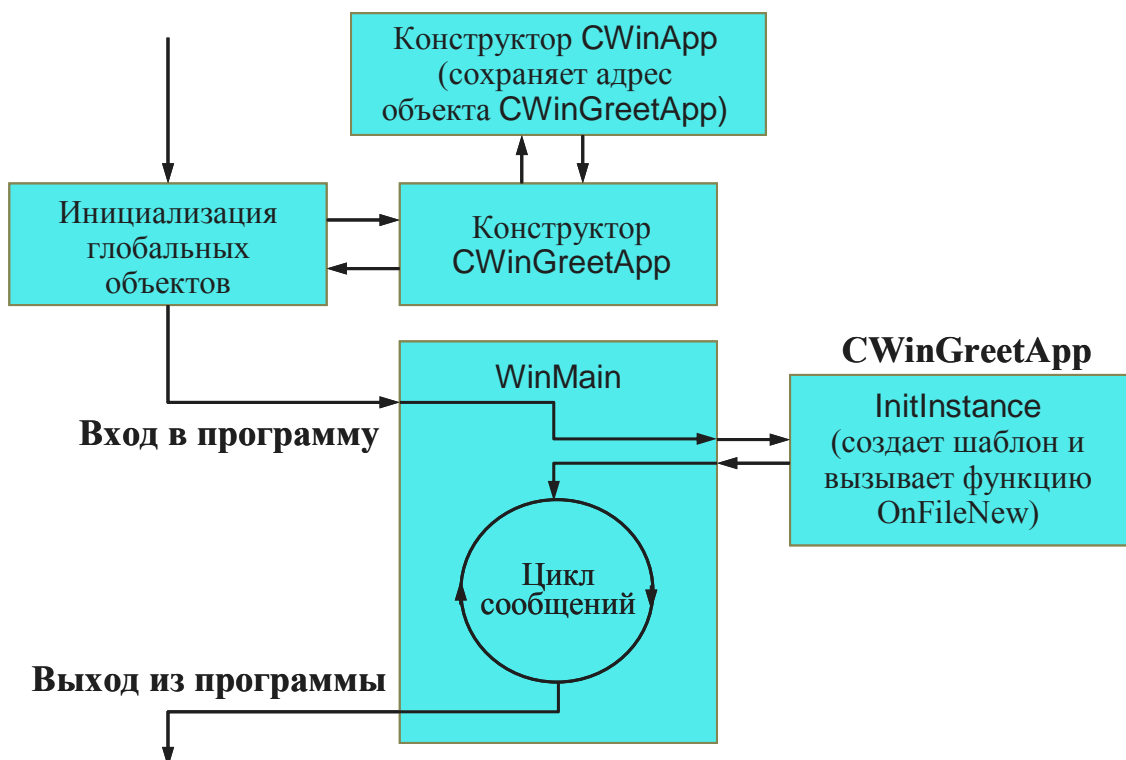
В файле WinGreet.cpp содержится единственный глобальный экземпляр класса CWinGreetApp:

```
CWinGreetApp theApp;
```

Таким образом, конструктор класса вызывается перед тем, как функция WinMain получает управление.

Конструктор CWinApp, вызываемый конструктором CWinGreetApp, выполняет две важные задачи:

- проверяет, объявлен ли в программе только один объект приложения (класса, порожденного от CWinApp),
- сохраняет адрес объекта в глобальном указателе, объявляемом в библиотеке MFC. После этого MFC получает возможность вызывать функции класса CWinGreetApp.



Получение управления функцией WinMain

Определена внутри MFC, подсоединяется при построении выполняемого файла, решает ряд задач, в частности, вызывает функцию InitInstance (виртуальная функция класса CWinApp), используя сохраненный адрес объекта CWinGreetApp.

Функция InitInstance служит для инициализации приложения.

Обработка сообщений функцией WinMain

После завершения задач инициализации функция WinMain входит в цикл для получения и распределения всех сообщений, посланных объектам внутри программы. В процессе работы приложения управление остается внутри этого цикла.

Выход из функции WinMain и завершение программы

Когда пользователь выбирает пункт меню File/Exit, системное меню/Close или нажимает кнопку Close, MFC уничтожает окно программы и вызывает функцию `PostQuitMessage` для выхода из цикла обработки сообщений. Вслед за этим происходит выход из функции `WinMain`, а следовательно, и приложения.

Функция InitInstance

Обычно в традиционных графических приложениях при вызове `InitInstance` создается только главное окно программы. Однако мастер `AppWizard` создает еще и шаблон документа. Шаблон – это экземпляр класса `CSingleDocTemplate` (в нашем случае).

Шаблон документа содержит:

- информацию о классах документа, главного окна и представления,
- идентификатор ресурсов программы (меню, значки и т.п.)

Если изучить исходный код функции `InitInstance`, то можно увидеть, что в нем выполняется следующая последовательность действий:

- создается указатель `pDocTemplate`;
- используется оператор `new` для динамического создания шаблона документа. Конструктору передаются идентификатор ресурсов программы и указатели на основные классы программы, при этом используется макрос `RUNTIME_CLASS`, возвращающий указатель на класс `CRuntimeClass`.
- указатель на объект шаблона передается в функцию `AddDocTemplate`.

После создания шаблона документа `InitInstance` путем вызова функции `ParseCommandLine` извлекает командную строку, а затем вызывает функцию `ProcessShellCommand` класса `CWinApp` для ее обработки. Так, если она содержит имя файла, то функция `ProcessShellCommand` пробует его открыть (в нашем случае пока не сработает, т.к. мы не написали соответствующий код). Если имени файла нет, то вызывается функция `OnFileNew` класса `CWinApp` для создания нового пустого документа.

При вызове функции `OnFileNew` программа использует шаблон документа для создания объектов типа `CWinGreetDoc`, `CMainFrame`, `CWinGreetView`, а также для связи с главным окном и окном представления. Они будут использоваться (в SDI-приложении), и при повторных вызовах `OnFileNew` новые объекты создаваться не будут.

Наконец, чтобы отобразить окно и его содержимое на экране, вызываются функции `ShowWindow` и `UpdateWindow` объекта главного окна.

Тема 6. Реализация представления

Представление (View) – часть программы с использованием библиотеки MFC для управления окном просмотра, обработки информации, вводимой пользователем, и отображением документа в окне.

Мы рассмотрим две реализации представления:

Графическое представление – простейшая программа рисования MiniDraw.

Текстовое представление – простейший текстовый редактор MiniEdit.

6.1. Реализация графического представления

Генерация исходных файлов

Повторить все шаги генерации программы WinGreet. В качестве имени проекта взять MiniDraw.

Определение переменных класса представления

В файле MiniDrawView.h добавить в описание класса CMiniDrawView переменные:

```
class CMiniDrawView : public CView
{
protected:
    CString m_ClassName;
    int m_Dragging;
    HCURSOR m_HCross; // Дескриптор указателя мыши
    CPoint m_PointOld;
    CPoint m_PointOrigin;
    // оставшаяся часть определения CMiniDrawView
}
```

Инициализация переменных класса представления

В файле MiniDrawView.cpp добавить в конструктор класса CMiniDrawView код инициализации переменных m_Dragging и m_HCross.

```
////////////////////////////////////
// CMiniDrawView construction/destruction

CMiniDrawView::CMiniDrawView()
{
    // TODO: добавьте код конструктора
    m_Dragging = 0;
    m_HCross = AfxGetApp() ->LoadStandardCursor (IDC_CROSS);
}
```


Переменная `m_HCross` содержит дескриптор указателя мыши, когда тот находится внутри окна представления. Функция `AfxGetApp` возвращает указатель на объект класса приложения (класс `CMiniDrawApp`). Он используется для получения через функцию `LoadStandardCursor` дескриптора стандартного крестообразного указателя.

Идентификаторы стандартных указателей Windows, которые можно передавать функции `LoadStandardCursor`

Значение	Вид указателя
<code>IDC_ARROW</code>	Стандартный указатель-стрелка
<code>IDC_CROSS</code>	Перекрестие, используемое для выбора
<code>IDC_BEAM</code>	Указатель для редактирования текста
<code>IDC_SIZEALL</code>	Указатель из четырех стрелок для изменения размеров окна
<code>IDC_SIZENESW</code>	Двунаправленная стрелка (северо-восток – юго-запад)
<code>IDC_SIZENS</code>	Двунаправленная стрелка (север – юг)
<code>IDC_SIZENWSE</code>	Двунаправленная стрелка (северо-запад – юго-восток)
<code>IDC_SIZEWE</code>	Двунаправленная стрелка (запад – восток)
<code>IDC_UPARROW</code>	Вертикальная стрелка
<code>IDC_WAIT</code>	"Песочные часы"

Добавление обработчиков сообщений Windows

С каждым окном связана функция, называемая процедурой окна. Операционная система вызывает эту функцию, если происходит событие, требующее обработки, и передает ей идентификатор события и данные для его обработки. Это называется передачей сообщения окну.

Процедура окна с помощью библиотеки MFC создается автоматически и выполняет минимальную стандартную обработку переданного сообщения. Если необходима собственная обработка сообщения, то должна быть создана соответствующая функция, являющаяся членом класса управления окном.

Для определения обработчика сообщения можно воспользоваться `ClassWizard`.

Командные сообщения

MFC обеспечивает специальную обработку сообщений, генерируемых объектами пользовательского интерфейса: меню, комбинации клавиш, кнопки панелей инструментов, строки состояния, элементы управления диалоговых окон.

Эти сообщения принято называть командными сообщениями.

Когда пользователь выбирает объект интерфейса или один из этих объектов необходимо обновить, объект передает командное сообщение главному окну.

Однако MFC сначала направляет это сообщение объекту окна представления. Если тот не имеет своего обработчика, то сообщение направляется объекту документа. Если объект документа не содержит обработчик, то сообщение передается главному окну программы. Если оно не располагает своим обработчиком, сообщение направляется объекту приложения. Наконец, если он не обеспечивает обработку, то сообщение обрабатывается стандартным образом.

Такое расширение основного механизма сообщений позволяет осуществлять обработку командных сообщений не только объектами, управляющими окнами, но и любыми объектами приложения. Они в качестве предка имеют класс CCmdTarget, реализующего механизм передачи сообщений.

Таким образом, программист обрабатывает сообщения внутри класса, наиболее подходящего для этого в каждом конкретном случае. Например, сообщение о выборе пункта меню File/Exit разумно обрабатывать в классе приложения, а File/Save – в классе документа.

Пример: добавление обработчика нажатия левой кнопки мыши

Добавим в программу функцию обработки сообщения WM_LBUTTONDOWN, которое передается при каждом нажатии левой кнопки мыши. Требуется:

- Открыть проект MiniDraw. Выбрать пункт меню View/ClassWizard или Ctrl-W.
- Открыть вкладку Message Maps, чтобы определить обработчик событий.
- В списке Class name выбрать класс CMiniDrawView (класс представления).
- В списке Object IDs выбрать пункт CMiniDrawView. Это позволит задать функцию-член класса CMiniDrawView для обработки любого уведомляющего сообщения, переданного классу представления, переопределив одну из виртуальных функций, которую он наследует от класса CView. Остальные пункты в списке – сообщения от объектов интерфейса.
- В списке сообщений Messages выбрать WM_LBUTTONDOWN. Список содержит идентификаторы всех уведомляющих сообщений и имена виртуальных функций, принадлежащих классу CView.
- Нажать кнопку AddFunction. ClassWizard создает шаблон обработчика сообщения с именем OnLButtonDown.

- Нажать кнопку Edit Code. Откроется файл MiniDrawView.cpp для редактирования кода функции OnLButtonDown.

Программирование обработки нажатия левой кнопки мыши

Добавим в функцию обработки OnLButtonDown следующий код:

```
void CMiniDrawView::OnLButtonDown(UINT nFlags, CPoint point)
{
    // TODO: Добавьте собственный код обработчика
    m_PointOrigin = point; // Начало линии
    m_PointOld = point;     // Используют другие обработчики
    SetCapture ();         // "Захват" сообщений от мыши окном
    m_Dragging = 1;        // Идет рисование
    RECT Rect;
    GetClientRect (&Rect); // Координаты окна представления
    ClientToScreen (&Rect); // Преобразуем в экранные
                           // (от верхнего левого угла)
    ::ClipCursor (&Rect); // Огр.перемещ.курс.пределами окна
    CView::OnLButtonDown(nFlags, point);
}
```

Схема сообщений

Схема сообщений (message map) – специальная структура MFC, описывающая связь функций с обрабатываемыми ими сообщениями. При создании функции-обработчика ClassWizard заносит ее в данную структуру (CMiniDrawView.h):

```
// Generated message map functions
protected:
    //{{AFX_MSG(CMiniDrawView)
    afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
    afx_msg void OnMouseMove(UINT nFlags, CPoint point);
    afx_msg void OnLButtonUp(UINT nFlags, CPoint point);
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
```

В файле реализации CMiniDrawView.cpp будут добавлены макросы:

```
BEGIN_MESSAGE_MAP(CMiniDrawView, CView)
    //{{AFX_MSG_MAP(CMiniDrawView)
    ON_WM_LBUTTONDOWN()
    ON_WM_MOUSEMOVE()
    ON_WM_LBUTTONUP()
    //}}AFX_MSG_MAP
    END_MESSAGE_MAP()
```

Схема сообщений – функция OnMouseMove

Так же как и ранее добавим обработчик события WM_MOUSEMOVE:

```
void CMiniDrawView::OnMouseMove(UINT nFlags, CPoint point)
{
    // TODO: Добавьте собственный код обработчика
    ::SetCursor (m_HCross); // Установка типа курсора
    if (m_Dragging)
    {
        CClientDC ClientDC (this);
        ClientDC.SetROP2 (R2_NOT);
        ClientDC.MoveTo (m_PointOrigin);
        ClientDC.LineTo (m_PointOld); // Стирание линии
        ClientDC.MoveTo (m_PointOrigin);
        ClientDC.LineTo (point); // Рисование новой линии
        m_PointOld = point;
    }
    CView::OnMouseMove(nFlags, point);
}
```

Вызов ClientDC.SetROP2 (R2_NOT) задает рисование в режиме инвертирования (при втором проходе линия стирается).

Схема сообщений – функция OnLButtonUp

Так же как и ранее добавим обработчик события WM_LBUTTONDOWN:

```
void CMiniDrawView:: OnLButtonUp(UINT nFlags, CPoint point)
{
    // TODO: Добавьте собственный код обработчика
    if (m_Dragging)
    {
        m_Dragging = 0;
        ::ReleaseCapture (); // Отменить захват сообщений мыши
        ::ClipCursor (NULL); // Курсор движется по всему экрану
        CClientDC ClientDC (this);
        ClientDC.SetROP2 (R2_NOT);
        ClientDC.MoveTo (m_PointOrigin);
        ClientDC.LineTo (m_PointOld);
        ClientDC.SetROP2 (R2_COPYPEN);
        ClientDC.MoveTo (m_PointOrigin);
        ClientDC.LineTo (point);
    }
    CView::OnLButtonUp(nFlags, point);
}
```

Параметры сообщений мыши

Всем обработчикам сообщений передаются два параметра: `nFlags` и `point`. Параметр `point` задает координаты острия курсора мыши. Отдельные биты параметра `nFlags` позволяют выяснить, были ли нажаты некоторые клавиши.

Битовая маска	Содержание бита устанавливается, если нажата
<code>MK_CONTROL</code>	клавиша Ctrl
<code>MK_LBUTTON</code>	левая кнопка мыши
<code>MK_MBUTTON</code>	средняя кнопка мыши
<code>MK_RBUTTON</code>	правая кнопка мыши
<code>MK_SHIFT</code>	клавиша Shift

Пример проверки, нажата ли клавиша Shift:

```
if (nFlags & MK_SHIFT)
```

Проектирование ресурсов программы

Произведем настройку ресурсов (меню и значок).

В окне Project Workspace откроем вкладку Resource View и развернем граф ресурсов программы (все категории).

Настройка меню:

- Двойной щелчок на `IDR_MAINFRAME` в разделе Menu (этот идентификатор используется также для настройки таблицы горячих клавиш и главного значка программы).
- Оставить в меню File только пункты New, Exit, а также разделитель.
- Убрать меню Edit.

Настройка значка:

- Двойной щелчок на `IDR_MAINFRAME` в разделе Icon. Файл изображения содержит две версии значка: крупный (32*32) и мелкий (16*16).
- Отредактировать (если надо).

Сохранение: пункт Save All меню File, основная информация о ресурсах сохранится в файле `MiniDraw.rc`, а информация о значке в `res\MiniDraw.ico`.

Настройка окна программы

Программа в настоящем виде имеет два проблемных момента:

1. Мерцание курсора. Причина: мы отображаем крестообразный, а Windows пытается отобразить обычную стрелку.

2. Если выбрать в панели управления темный цвет для окна, то линий не видно.

Решение: переписать виртуальную функцию `PreCreateWindow` класса `CView`:

```
BOOL CMiniDrawView::PreCreateWindow(CREATESTRUCT& cs)
{
    // TODO: Здесь измените класс или стили окна,
    // модифицируя структуру CREATESTRUCT cs

    m_ClassName = AfxRegisterWndClass
        (CS_HREDRAW | CS_VREDRAW,           // стили окна
        0,                                  // без курсора
        (HBRUSH)::GetStockObject (WHITE_BRUSH), // белый фон
        0);                                 // без значка
    cs.lpszClass = m_ClassName;

    return CView::PreCreateWindow(cs);
}
```

Поле `lpszClass` структуры `CREATESTRUCT` хранит имя *класса окна Windows* (это не класс C++, а структура данных для описания общих свойств окна).

Для регистрация класса используется глобальная функция библиотеки MFC `AfxRegisterWndClass`, имеющая следующие параметры:

1. Стили `CS_HREDRAW` и `CS_VREDRAW`, требующие перерисовки окна при изменении каждого из его размеров.
2. Задаёт форму указателя. Если 0, то *Windows* не отображает указатель. Здесь можно было сразу задать требуемый указатель курсора, однако мы хотим оставить себе возможность его оперативного изменения.
3. Заливка окна представления. В данном случае – стандартная белая кисть.
4. Значок окна. 0, если не нужен.

Обратите внимание на переопределения виртуальных функций, сгенерированные мастерами `AppWizard` и `ClassWizard` для определения схемы сообщений класса представления – см. определение класса `CMiniDrawView` в файле `CMiniDrawView.h`.

6.2. Реализация текстового представления

Простейший пример – создание текстового редактора (программа `MiniEdit`).

Для этого достаточно породить класс представления не от `CView`, а от `CEditView`. Это позволит вводить и редактировать текст внутри окна представления и использовать набор готовых команд редактирования.

Другие классы представления, реализованные в библиотеке MFC: CCtrlView, CFormView, CScrollView, CDaoRecordView, CHtmlView, CListView, COleDBRecordView, CRecordView, CRichEditView, CTreeView.

Генерация исходных файлов

1. Повторить все шаги генерации программы WinGreet. В качестве имени проекта взять MiniEdit.
2. При этом на шаге 6:
 - Выбрать имя класса CMiniEditView из списка в верхней части окна.
 - В списке Base class выбрать CEditView.
3. Отредактировать меню и таблицу горячих клавиш, как предложено ниже.

Редактирование ресурсов программы

1. Удалить в меню File все, кроме команды Exit и разделителя над ней.
2. Добавить новый пункт меню введя идентификатор ID_FILE_PRINT в поле ID и строку &Print...\tCtrl+P в поле Caption.
3. Перетащить мышью новую команду выше разделителя.
4. В меню Edit после пункта Paste вставить следующие пункты меню:
 - ID: ID_EDIT_SELECT_ALL, Caption: Select &All;
 - Разделитель (отметить опцию Separator)
 - ID: ID_EDIT_FIND, Caption: &Find;
 - ID: ID_EDIT_REPEAT, Caption: Find &Next\tF3;
 - ID: ID_EDIT_REPLACE, Caption: Replace;Обратите внимание: в нижней части окна Menu Item Properties отображается подсказка (выводится в строке состояния, если она есть).
5. Отредактировать таблицу горячих клавиш.

Редактирование таблицы горячих клавиш

Добавим в таблицу горячих клавиш вызовы команды печати и повторения поиска, заявленные ранее в меню.

1. Двойной щелчок на IDR_MAINFRAME в разделе Accelerator.
2. Добавить новый пункт двойным щелчком в пустой строке внизу таблицы.
3. Ввести идентификатор ID_FILE_PRINT в поле ID.
4. Нажать кнопку Next Key Typed, затем комбинацию Ctrl+P.
5. Аналогичным образом назначить идентификатору ID_EDIT_REPEAT клавишу F3.

Если необходимо, можно отредактировать значки программы, затем сохранить весь проект.

Собрать проект, запустить на выполнение. Обратить внимание, что все команды меню уже реализованы, так как обрабатываются средствами класса CEditView.

Тема 7. Реализация документа

Класс документа в архитектуре Document/View отвечает за хранение данных, за их загрузку (и сохранение) из файлов, а также содержит функции, позволяющие другим классам получать или изменять данные, осуществляя таким образом их просмотр и редактирование.

В данной теме будут рассмотрены следующие вопросы:

- Сохранение графических данных (на примере программы MiniDraw)
- Перерисовка окна
- Добавление в меню команд Undo и Delete All
- Удаление данных из документа
- Реализация команд меню Undo и Delete All

7.1. Сохранение графических данных

Определение класса для сохранения информации о введенных линиях

Добавим в файле MiniDrawDoc.h перед определением класса CMiniDrawDoc описание класса CLine:

```
class CLine : public CObject
{
protected:
    int m_X1, m_Y1, m_X2, m_Y2;
public:
    CLine(int X1, int Y1, int X2, int Y2)
    {
        m_X1 = X1;
        m_Y1 = Y1;
        m_X2 = X2;
        m_Y2 = Y2;
    }
    void Draw(CDC *PDC);
};
```

Дополнения в классе документа

Добавим в коде определения класса CMiniDrawDoc (файл MiniDrawDoc.h):

```
class CMiniDrawDoc : public CDocument
{
protected:
    CTypedPtrArray <CObArray, CLine*> m_LineArray;
public:
    void AddLine(int X1, int Y1, int X2, int Y2);
```

```

    CLine *GetLine(int Index);
    int GetNumLines();
// Остальные определения класса CMiniDrawDoc ...
}

```

Здесь `m_LineArray` – экземпляр шаблона MFC-класса `CTypedPtrArray`. Этот класс генерирует семейство классов, каждый из которых является производным от класса, указанного в первом параметре шаблона (может быть `CObArray` или `CPtrArray`). Каждый из классов предназначен для хранения переменных класса, указанного во втором параметре шаблона.

Таким образом, переменная `m_LineArray` является объектом класса, порожденного от класса `CObArray` и сохраняющего указатели на `CLine`.

Для использования шаблона класса `CTypedPtrArray` (как и любого другого шаблона) в программу надо включить (лучше в стандартный файл заголовков `StdAfx.h`) файл `Afxtempl.h`:

```

#include <afxtempl.h>           // Шаблоны библиотеки MFC

```

Реализация функций класса документа

В конце файла реализации документа `MiniDrawDoc.cpp` добавим код функций `Draw`, `AddLine`, `GetLine`, `GetNumLines`:

```

void CLine::Draw(CDC *PDC)
{
    PDC->MoveTo(m_X1, m_Y1);
    PDC->LineTo(m_X2, m_Y2);
}

void CMiniDrawDoc::AddLine(int X1, int Y1, int X2, int Y2)
{
    CLine *PLine = new CLine(X1, Y1, X2, Y2);
    m_LineArray.Add(PLine);
}

CLine *CMiniDrawDoc::GetLine(int Index)
{
    if (Index < 0 || Index > m_LineArray.GetUpperBound())
        return 0;
    return m_LineArray.GetAt(Index);
}

int CMiniDrawDoc::GetNumLines()
{
    return m_LineArray.GetSize();
}

```

Внесем необходимые добавления в код функции OnLButtonUp (файл MiniDrawView.cpp):

```
void CMiniDrawView:: OnLButtonUp(UINT nFlags, CPoint point)
{
    // ...
    ClientDC.LineTo (point);
    CMiniDrawDoc* PDoc = GetDocument();
    PDoc->AddLine (m_PointOrigin.x, m_PointOrigin.y,
                  point.x, point.y); // Запомнить линию
    // ...
}
```

7.2. Перерисовка окна

Поскольку теперь программа хранит данные, позволяющие восстановить линию, их можно использовать при перерисовке окна.

Для перерисовки окна система сначала удаляет его содержимое, а затем вызывает функцию OnDraw класса представления.

Добавим в файл MiniDrawView.cpp код:

```
void CMiniDrawView::OnDraw(CDC* pDC)
{
    CMiniDrawDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);

    // TODO: добавьте код отображения собственных данных
    int Index = pDoc->GetNumLines();
    while (Index-->0)
        pDoc->GetLine (Index) ->Draw(pDC);
}
```

Замечание: система удаляет только ту часть окна, которую требуется перерисовать, поэтому процесс отрисовки можно еще оптимизировать.

7.3. Добавление команд в меню

Для добавления необходимых пунктов в меню выполним следующие действия:

1. Откроем редактор главного меню на вкладке Resource View окна Workspace.
2. Добавим пункт Edit в строке меню. (Caption: &Edit). Перетащим его между пунктами File и Help.
3. Добавим пункты меню в меню Edit:
 - ID: ID_EDIT_UNDO, Caption: &Undo\tCtrl+Z;
 - Разделитель (отметить опцию Separator)
 - ID: ID_EDIT_CLEAR_ALL, Caption: &Delete All;
4. Сохраним проект.

Замечания: для команды Undo не требуется определять комбинацию клавиш Ctrl+Z, так как AppWizard определил ее при первичной генерации кода программы. Кроме того, он назначил для нее комбинацию Alt+Backspace (обычную для ранних графических приложений).

При использовании редактора меню можно задавать некоторые дополнительные установки пунктов меню, например, сделать пункт изначально недоступным.

7.4. Удаление данных документа

Когда выбирается пункт New меню File, MFC (а точнее функция OnFileNew класса CWinApp) вызывает виртуальную функцию CDocument::DeleteContents. Чтобы удалить данные, сохраняемые нашим классом документа (CMiniDrawDoc), необходимо написать собственную реализацию этой функции. Для этого:

1. Вызвать мастера ClassWizard (например, Ctrl+W). Открыть вкладку Message Map, позволяющую определить функции-члены.
2. В списке Class name выбрать класс CMiniDrawDoc. В списке Object IDs выбрать CMiniDrawDoc для отображения в списке Messages имен виртуальных функций, определенных в родительских классах.

В списке Messages выбрать функцию DeleteContents, нажать кнопки Add Function, затем Edit Code. Добавить в функцию следующие строки:

```
void CMiniDrawDoc::DeleteContents()
{
    // TODO: Добавьте собственный код обработчика
    int Index = m_LineArray.GetSize();
    while (Index--)
        delete m_LineArray.GetAt(Index);
    m_LineArray.RemoveAll();
    CDocument::DeleteContents();
}
```

7.5. Реализация команд меню

Обработка команды Delete All

1. Вызвать мастера ClassWizard (Ctrl+W). Открыть вкладку Message Map.
2. В списке Class name выбрать класс CMiniDrawDoc, чтобы функция стала членом этого класса (командное сообщение может обрабатываться любым из четырех основных классов: представления, документа, главного окна или приложения).
3. В списке Object IDs выбрать идентификатор ID_EDIT_CLEAR_ALL. Именно он при создании меню был присвоен команде Delete All. После этого в списке Messages появятся идентификаторы двух воз-

можных типов сообщений: `COMMAND` и `UPDATE_COMMAND_UI`. Первый означает, что пользователь выбрал этот пункт меню, второй соответствует сообщению, передаваемому при первом открытии меню, содержащего команду.

4. Выбрать сообщение `COMMAND`.
5. Нажать кнопку `Add Function`, согласиться с предлагаемым стандартным именем функции `OnEditClearAll`.
6. Нажать кнопку `Edit Code`.
Добавить в функцию следующий код:

```
void CMiniDrawDoc::OnEditClearAll()
{
    // TODO: Добавьте собственный код обработчика
    DeleteContents();
    UpdateAllViews(0); //Удалить содержимое окна представления
}
```

7. Выполнить шаги с 4 по 7 для добавления обработчика сообщения `UPDATE_COMMAND_UI`. Ввести код:

```
void CMiniDrawDoc::OnUpdateEditClearAll(CCmdUI* pCmdUI)
{
    // TODO: Добавьте собственный код обработчика
    pCmdUI->Enable(m_LineArray.GetSize());
}
```

Функция `OnUpdateEditClearAll` получает указатель на объект класса `CCmdUI`, предоставляющий функции для инициализации команд меню и других объектов пользовательского интерфейса. Так, функция `Enable` делает доступной команду `Delete All`, если документ не пуст. Другие функции: `SetCheck`, `SetRadio`, `SetText`.

Обработка команды *Undo*

С помощью мастера `ClassWizard` создать обработчик команды `Undo`, идентификатор: `ID_EDIT_UNDO`, имя функции: `OnEditUndo`. Код:

```
void CMiniDrawDoc::OnEditUndo()
{
    // TODO: Добавьте собственный код обработчика
    int Index = m_LineArray.GetUpperBound();
    if (Index > -1)
    {
        delete m_LineArray.GetAt(Index);
        m_LineArray.RemoveAt(Index);
    }
    UpdateAllViews(0);
}
```

Добавить функцию инициализации команды меню Undo. Ее действие распространяется в том числе и на нажатие горячих клавиш.

```
void CMiniDrawDoc::OnUpdateEditUndo(CCmdUI* pCmdUI)
{
    // TODO: Добавьте собственный код обработчика
    pCmdUI->Enable(m_LineArray.GetSize());
}
```

Тема 8. Хранение данных

Тема посвящена изучению принципов сохранения и загрузки данных документа из файлов на диске.

В рамках данной темы мы реализуем стандартные команды меню File: New, Open, Save, Save As, – а также технологию drag-and-drop (открытие файла путем перетаскивания объекта файла из папки Windows или окна Windows Explorer в окно программы).

Мы рассмотрим в качестве примеров следующие действия:

- Добавление средств ввода-вывода данных в программу MiniDraw
- Добавление средств ввода-вывода данных в программу MiniEdit
- Другие средства ввода-вывода файлов

8.1. Ввод-вывод программы MiniDraw

Добавление команд в меню File

В редакторе меню добавить ниже пункта New команды Open..., Save, Save As..., разграничитель и команду Recent File:

Идентификатор	Надпись	Другие свойства
ID_FILE_OPEN	&Open...\tCtrl+O	–
ID_FILE_SAVE	&Save\tCtrl+S	–
ID_FILE_SAVE_AS	Save &As...	–
–	–	Separator
ID_FILE_MRU_FILE1	Recent File	Grayed

Замечание:

Если в программе открыт хотя бы один файл, MFC вставит его имя вместо пункта Recent File, а также добавит имена последних использованных файлов (не более 4).

Задание стандартного расширения файлов

В существующей программе:

- В редакторе строковых ресурсов двойным щелчком открыть строку с идентификатором IDR_MAINFRAME.
- В поле Caption ввести необходимую информацию (не нажимая Enter!). Здесь: **MiniDraw\n\nMiniDr\nMiniDraw Files (*.drw)\n.drw\n Mini-Draw.Document\n MiniDr Document**

Информация	Что описано
MiniDraw\n	Заголовок окна приложения
\n	Основа для имен документа. Если нет – Untitled
MiniDr\n	Имя типа документа
MiniDraw Files (*.drw)\n	Описание типа документа и фильтр
.drw\n	Расширение документов этого типа
MiniDraw.Document\n	Идентификатор типа файлов в реестре
MiniDr Document	Описание типа файлов в реестре

На этапе создания программы:

При создании приложения посредством AppWizard:

1. В диалоговом окне AppWizard (Step 4) нажать кнопку Advanced... Откроется диалоговое окно Advanced Options, выбрать в нем вкладку Document Template Strings.
2. В поле File extension задать стандартное расширение файла (без точки), например, drw.
3. AppWizard сформирует описание расширения в поле Filter name. При желании эту строку можно отредактировать.
4. Нажать кнопку Close и продолжить создание приложения.

Поддержка команд меню File

Для команд New, Open..., Save и Save As... определять обработчики не требуется, так как они предоставляются MFC. В этом случае необходимо написать код их *поддержки*. MFC также предоставляет обработчики команд для работы с последними использованными файлами в меню File.

Команда New обрабатывается функцией CWinApp::OnFileNew. Она вызывает виртуальную функцию DeleteContents, а затем инициализирует новый документ.

Команда Open... обрабатывается функцией CWinApp::OnFileOpen. Она отображает стандартное диалоговое окно Open. Если выбрать файл и нажать кнопку Open, OnFileOpen откроет файл для чтения, а затем вызовет функцию CMiniDrawDoc::Serialize, которая и должна обеспечить чтение данных.

Команда Save обрабатывается функцией CDocument::OnFileSave, а *Save As* – функцией CDocument::OnFileSaveAs.

OnFileSave (при первом сохранении документа) и OnFileSaveAs открывают диалоговое окно Save As, затем (если задано имя файла и нажата кнопка Save) открывают файл на запись и вызывают CMiniDrawDoc::Serialize для выполнения собственно операций записи.

Сериализация данных документа

Минимальная реализация функции `CMiniDrawDoc::Serialize` была создана мастером `AppWizard`. Добавим в нее следующий код:

```
void CMiniDrawDoc::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
    {
        // TODO: здесь добавьте код сохранения данных
        m_LineArray.Serialize (ar);
    }
    else
    {
        // TODO: здесь добавьте код загрузки данных
        m_LineArray.Serialize (ar);
    }
}
```

Функция `CArchive::IsStoring` возвращает `TRUE`, если файл был открыт для записи (выбрана команда `Save` или `Save As`) и `FALSE` – если для чтения (команда `Open`).

Переменная `m_LineArray` имеет собственную функцию-член `Serialize` (наследуемую от класса `CObArray`), которая вызывается для чтения и записи всех объектов класса `CLine`, хранимых в данной переменной.

Функция `CObArray::Serialize` выполняет для каждого объекта класса `CLine` следующие действия:

при записи:

1. Записывает в файл информацию о классе объекта
2. Вызывает функцию `Serialize` объекта для записи данных.

при чтении:

1. Считывает информацию из файла, динамически создает объект и сохраняет указатель на него.
2. Вызывает функцию `Serialize` объекта для чтения данных.

Теперь, чтобы обеспечить поддержку сериализации класса `CLine`, требуется включить в его определение (в `MiniDrawDoc.h`) макрос `DECLARE_SERIAL` и определить конструктор класса по умолчанию:

```
class CLine : public CObject
{
protected:
    int m_X1, m_Y1, m_X2, m_Y2;
    CLine () {}
    DECLARE_SERIAL (CLine)
// ...
```

В файл MiniDrawDoc.cpp требуется включить макрос
IMPLEMENT_SERIAL:

```
////////////////////////////////////  
// CMiniDrawDoc commands  
IMPLEMENT_SERIAL(CLine,CObject,1) // Класс, баз.класс, версия
```

Последнее, что нужно сделать – это добавление функции Serialize для класса CLine, а именно в описании класса (в MiniDrawDoc.h) ее определение:

```
public:  
    virtual void Serialize (CArchive& ar);
```

и в файле реализации (в MiniDrawDoc.cpp) – ее код:

```
void CLine::Serialize (CArchive& ar)  
{  
    if (ar.IsStoring())  
        ar << m_X1 << m_Y1 << m_X2 << m_Y2;  
    else  
        ar >> m_X1 >> m_Y1 >> m_X2 >> m_Y2;  
}
```

Замечание:

Класс, объекты которого должны быть сериализованы, должен прямо или косвенно порождаться от MFC-класса CObject.

В соответствии с принципами ООП для переменных, являющихся объектами класса, вызывается функция Serialize их собственного класса, для переменных, не являющихся объектами для сериализации, разумно использовать перегруженные операторы << и >> класса CArchive.

Установка флага изменений

Класс CDocument поддерживает флаг изменений, показывающий содержит ли документ не сохраненные данные. MFC проверяет этот флаг перед вызовом функции DeleteContents. Он устанавливает флаг в FALSE, когда документ открыт и сохранен. Для установки его надо вызвать функцию SetModifiedFlag класса CDocument (аргумент по умолчанию равен TRUE):

```
void CMiniDrawDoc::AddLine(int X1, int Y1, int X2, int Y2)  
{  
    // ...  
    SetModifiedFlag ();  
}  
void CMiniDrawDoc::OnEditClearAll()  
{  
    // ...  
    SetModifiedFlag ();  
}
```

```
void CMiniDrawDoc::OnEditUndo()
{
    // ...
    SetModifiedFlag ();
}
```

Поддержка технологии "drag-and-drop"

Для поддержки операции перетаскивания необходимо вызвать функцию `CWnd::DragAcceptFiles` для объекта главного окна. Это нужно сделать внутри функции `CMiniDrawApp::InitInstance` (в `MiniDraw.cpp`) после вызова функции `UpdateWindow`:

```
BOOL CMiniDrawApp::InitInstance()
{
    // ...
    m_pMainWnd->UpdateWindow();

    m_pMainWnd->DragAcceptFiles();
    return TRUE;
}
```

Объект приложения содержит переменную `m_pMainWnd`, являющуюся указателем на объект главного окна. Все вызовы его функций должны происходить после вызова функции `ProcessShellCommand`, так как внутри последней создается главное окно и присваивается значение переменной `m_pMainWnd`.

После вызова функции `DragAcceptFiles` (когда пользователь отпускает значок), MFC автоматически открывает файл, создает объект класса `CArchive` и вызывает функцию `Serialize`.

Регистрация типа файла

В системный реестр Windows полезно добавить информацию о том, что для открытия файлов с определенным расширением (здесь: `.drw`) следует использовать конкретную программу (`MiniDraw`). Для этого в функции `InitInstance` (`MiniDraw.cpp`) добавим вызовы функций `EnableShellOpen` и `RegisterShellFileTypes`:

```
BOOL CMiniDrawApp::InitInstance()
{
    // ...
    AddDocTemplate(pDocTemplate);

    EnableShellOpen();
    RegisterShellFileTypes();
    // ...
}
```

Замечание:

Обращения к функциям EnableShellOpen и RegisterShellFileTypes помещаются после вызова функции AddDocTemplate, добавляющей шаблон документа в приложение, чтобы информация о стандартном расширении файла и типе документа была доступна объекту приложения.

8.2. Ввод-вывод программы MiniEdit

Добавление команд в меню File

В редакторе меню добавить над пунктом Print команды New, Open..., Save и Save As..., а ниже пункта Print – разграничитель и команду Recent File:

Идентификатор	Надпись	Другие свойства
ID_FILE_NEW	&New\tCtrl+N	—
ID_FILE_OPEN	&Open...\tCtrl+O	—
ID_FILE_SAVE	&Save\tCtrl+S	—
ID_FILE_SAVE_AS	Save &As...	—
—	—	Separator
ID_FILE_MRU_FILE1	Recent File	Grayed

Изменить строковый ресурс с идентификатором IDR_MAINFRAME:

**MiniEdit\n\nMiniEd\nMiniEdit Files (*.txt)\n.txt\nMiniEdit.Document
\nMiniEd Document**

Добавление кода поддержки

В файле MiniEditDoc.cpp добавить строки в коде функции DeleteContents:

```
void CMiniEditDoc::DeleteContents()
{
    // TODO: Добавьте собственный код обработчика
    POSITION Pos = GetFirstViewPosition ();
    CEditView *PCEditView = (CEditView *)GetNextView (Pos);
    if (PCEditView)
        PCEditView->SetWindowText ("");
    CDocument::DeleteContents ();
}
```

Так как класс представления порожден от CEditView, текст сохраняется в самом окне представления, чтобы его удалить делается следующее:

- Через функцию GetFirstViewPosition получается индекс первого (единственного) объекта представления.

- Через функцию `GetNextView` получается указатель на класс представления документа (и инкрементирует свой аргумент).
- Если указатель получен, очищает содержимое документа.

В классе `CEditView` флаг изменений в нем уже реализован. Реализация технологии `drag-and-drop` выполняется так же, как и в программе `MiniDraw`.

Функции `Read` и `Write` класса `CArchive`

Если для ввода-вывода данных по какой-то причине неудобно использовать перегруженные операторы `<<` и `>>`, можно воспользоваться функциями `Read` и `Write` класса `CArchive`:

```
UINT Read( void* lpBuf, UINT nMax );  
void Write( const void* lpBuf, UINT nMax );
```

Здесь `lpBuf` – указатель на буфер для ввода-вывода данных,
`nMax` – количество читаемых или записываемых байт.

При этом не производится никакого форматирования данных и записи о классе в файл.

8.3. Другие средства ввода-вывода файлов

Сериализация обеспечивает удобный способ ввода-вывода, при котором данные сохраняются в двоичном формате и вместе с ними хранится информация о версии программы и классе.

В качестве альтернативы MFC позволяет использовать низкоуровневые методы ввода-вывода (перемещать файловый указатель, записывать и считывать определенное число байтов и т.п.).

Один из способов – использование класса `CFile`. Объект этого класса присоединяется к указанному файлу и позволяет выполнять обширный набор функций для выполнения универсальных операций двоичного ввода-вывода.

Можно присоединить такой объект и к уже открытому файлу – объекту `CArchive` (функция `CArchive::GetFile`).

Для ввода-вывода текстового файла можно использовать объекты класса `CStdioFile`. Также можно использовать класс `CMemFile`.

Тема 9. Прокрутка и разделение окон представления

Прокрутка делает возможным просмотр и редактирование документа, размеры которого превышают размеры окна представления.

Разделение позволяет создавать несколько окон представления одного документа и прокручивать его в каждом окне отдельно.

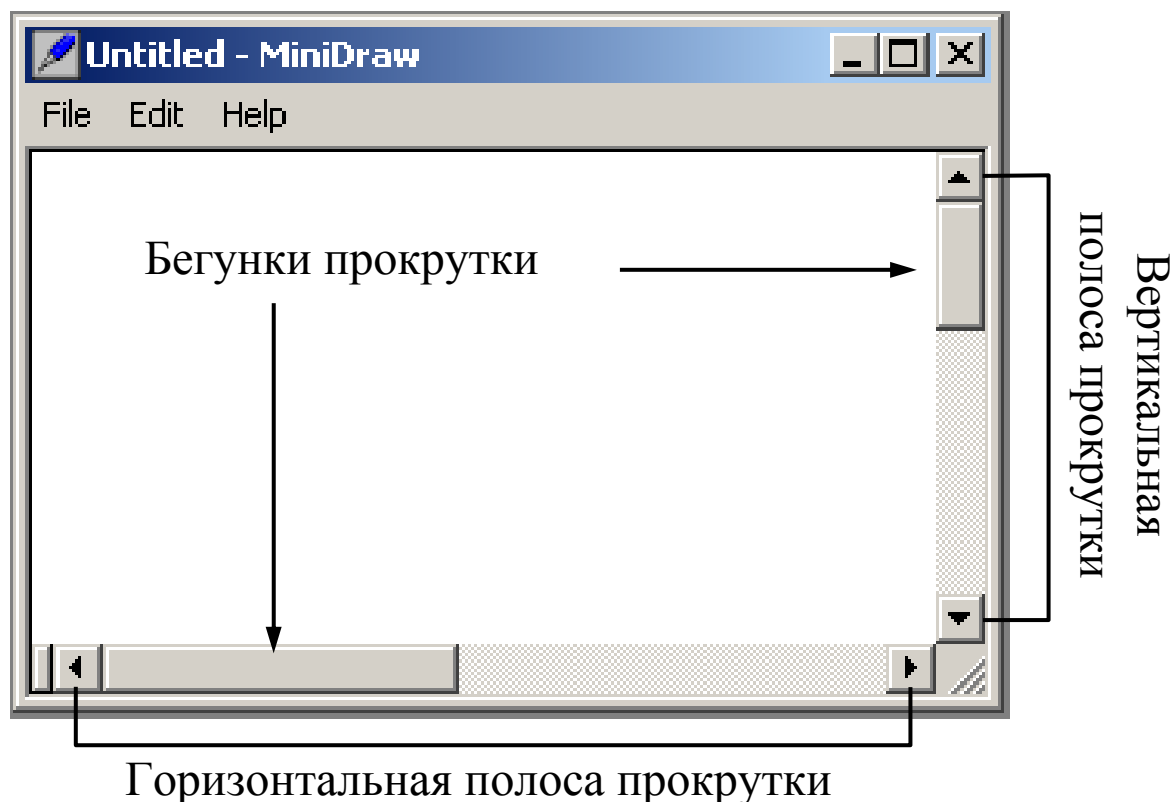
В качестве примера используется программа MiniDraw. В программе MiniEdit это делать вручную не требуется, так как класс CEditView, от которого порождается класс представления, делает все сам.

Ниже рассматриваются следующие вопросы:

- Добавление средств прокрутки окна
- Добавление средств разделения окна
- Обновление окна представления

9.1. Добавление средств прокрутки окна

В файлах MiniDrawView.h и MiniDrawView.cpp заменить базовый класс окна представления с CView на CScrollView. Результат:



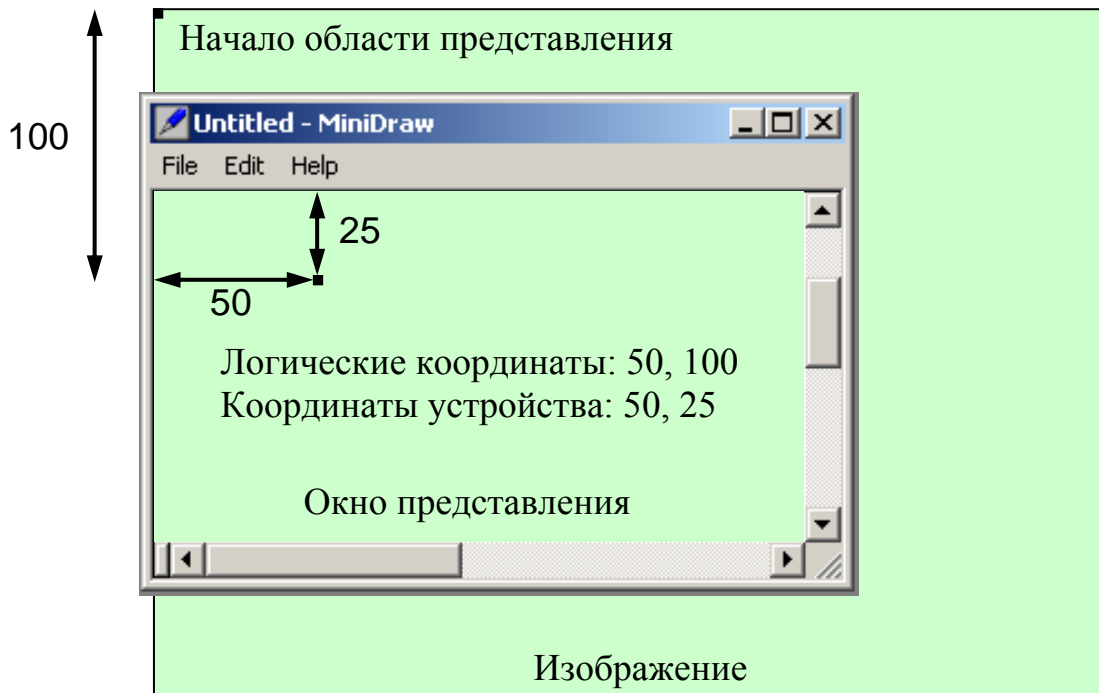
Замечание: Можно было выбрать класс CScrollView в качестве базового сразу при генерации кода программы посредством AppWizard.

Преобразование координат

Способ поддержки прокрутки в классе CScrollView:

При первом открытии документа его левый верхний угол отображается в левом верхнем углу окна. При прокрутке документа с помощью полосы прокрутки MFC корректирует значение атрибута, называемого началом области просмотра.

Например, если прокрутить документ на 75 пикселей вниз, то точка с координатами (50, 100) будет выведена на расстоянии 25 пикселей от верхней границы окна – см. рисунок.



После того, как MFC скорректирует начало области просмотра, функция OnDraw перерисует линии в окне представления, задавая те же самые координаты, так как функции рисования MFC (в данном случае MoveTo и LineTo) принимают логические координаты.

Однако при работе с мышью MFC использует координаты устройства. Поэтому в нашей программе после добавления средств прокрутки окна следует добавить код для *преобразования физических координат в логические*.

В функции OnLButtonDown (файл MiniDrawView.cpp) добавим следующий код:

```
void CMiniDrawView::OnLButtonDown(UINT nFlags, CPoint point)
{
    // TODO: Добавьте собственный код обработчика
    CClientDC ClientDC (this);
    OnPrepareDC (&ClientDC);
    ClientDC.DPtoLP (&point);
    m_PointOrigin = point;    // Начало линии
    // ...
}
```


Первый добавленный оператор создает объект контекста, относящийся к окну представления. Второй вызывает функцию `OnPrepareDC` класса `CScrollView`, корректирующую начало области просмотра на основании текущей позиции прокрученного рисунка. Третий вызывает функцию `DPTtoLP` класса `CClientDC`, преобразующую координаты из физических в логические.

Замечание: Объект контекста устройства, передаваемый функции `OnDraw`, уже имеет начало области просмотра, скорректированное для прокрученного рисунка. При создании собственного контекста устройства, его необходимо передать функции `OnPrepareDC` для коррекции начала области просмотра.

Аналогичным образом изменим код функций `OnMouseMove` и `OnLButtonUp`:

```
void CMiniDrawView::OnMouseMove(UINT nFlags, CPoint point)
{
    // ...
    if (m_Dragging)
    {
        CClientDC ClientDC (this);
        OnPrepareDC (&ClientDC);
        ClientDC.DPTtoLP (&point);
        ClientDC.SetROP2 (R2_NOT);
    }
    // ...
}

void CMiniDrawView::OnLButtonUp(UINT nFlags, CPoint point)
{
    if (m_Dragging)
    {
        // ...
        CClientDC ClientDC (this);
        OnPrepareDC (&ClientDC);
        ClientDC.DPTtoLP (&point);
        ClientDC.SetROP2 (R2_NOT);
    }
    // ...
}
```

Ограничение размера рисунка

Очевидно, для правильного отображения бегунков прокрутки (и не только для этого) MFC должна иметь информацию о размерах рисунка. Для этого мы переопределим виртуальную функцию `OnInitialUpdate` в классе представления программы. Сначала для создания минимального определения функции вызовем Class Wizard (Ctrl+W), выберем вкладку Message Map, выберем класс `CMiniDrawView` в списках Class name и Object IDs, а затем выберем виртуальную функцию `OnInitialUpdate` в спис-

ке Messages. Добавим код в определение функции в файле MiniDrawView.cpp:

```
void CMiniDrawView::OnInitialUpdate()
{
    CScrollView::OnInitialUpdate();
    // TODO: Добавьте собственный код обработчика
    SIZE Size = {640, 480};
    SetScrollSizes (MM_TEXT, Size);
}
```

Здесь мы установили размер рисунка постоянным и равным 640*480. Если его надо изменять по ходу работы программы, разумно это делать внутри функции OnUpdate класса представления (вызывается при каждом изменении документа).

Первый аргумент функции SetScrollSizes задает режим отображения (атрибут, сохраняемый объектом контекста устройства). Режим MM_TEXT означает, что размеры задаются в пикселях, а координаты растут слева направо и сверху вниз. Второй аргумент – структура SIZE. Через третий и четвертый аргумент можно передать размеры страницы и строки (используются при щелчках по полосе прокрутки).

Чтобы предотвратить рисование за пределами рисунка (если он меньше окна представления), изменим код функции OnDraw в файле MiniDrawView.cpp:

```
void CMiniDrawView::OnDraw(CDC* pDC)
{
    CMiniDrawDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    // TODO: добавьте код отображения собственных данных
    CSize ScrollSize = GetTotalSize(); // Получим размер
    pDC->MoveTo(ScrollSize.cx, 0);
    pDC->LineTo(ScrollSize.cx, ScrollSize.cy); // Рисуем
    pDC->LineTo(0, ScrollSize.cy); // границу
    int Index = pDoc->GetNumLines();
    // ...
}
```

Добавим операторы, предотвращающие рисование за пределами рисунка. Для этого изменим код функции OnLButtonDown в файле MiniDrawView.cpp:

```
void CMiniDrawView::OnLButtonDown(UINT nFlags, CPoint point)
{
    // ...
    ClientDC.DPtoLP (&point);
    // Находимся ли внутри области окна представления
    CSize ScrollSize = GetTotalSize ();
}
```

```

CRect ScrollRect (0, 0, ScrollSize.cx, ScrollSize.cy);
if (!ScrollRect.PtInRect (point)) // TRUE, если внутри
    return;
// Сохранение позиции, "захват" мыши, установка флага
// ...
// Ограничение перемещений указателя мыши
ClientDC.LPtoDP (&ScrollRect);
CRect ViewRect; // Координаты окна представления
GetClientRect (&ViewRect);
CRect IntRect; // Координаты пересечения области рисунка
IntRect.IntersectRect (&ScrollRect, &ViewRect);
ClientToScreen (&IntRect); // В координаты экрана
::ClipCursor (&IntRect); // Ограничить перемещение
CScrollView::OnLButtonDown(nFlags, point);
}

```

Изменение формы указателя

Добавим операторы для изменения формы указателя мыши на крестообразную внутри области рисунка (можно рисовать) и восстанавливающие обычную форму в виде стрелки за пределами этой области. Добавим переменную `m_HArrow` в определение класса `CMiniDrawView` в файле `MiniDrawView.h`:

```

class CMiniDrawView : public CScrollView
{
    // ...
    int m_Dragging;
    HCURSOR m_HArrow;
    HCURSOR m_HCross;
    // ...
}

```

Инициализируем переменную `m_HArrow` в конструкторе класса `CMiniDrawView` в файле `MiniDrawView.cpp`:

```

CMiniDrawView::CMiniDrawView()
{
    // TODO: добавьте код конструктора
    m_Dragging = 0;
    m_HArrow = AfxGetApp() ->LoadStandardCursor(IDC_ARROW);
    m_HCross = AfxGetApp() ->LoadStandardCursor(IDC_CROSS);
}

```

В функции `OnMouseMove` в файле `MiniDrawView.cpp` переместим объявление объекта `ClientDC` и вызовы методов `OnPrepareDC` и `DPtoLP` в начало функции и добавим следующий код:

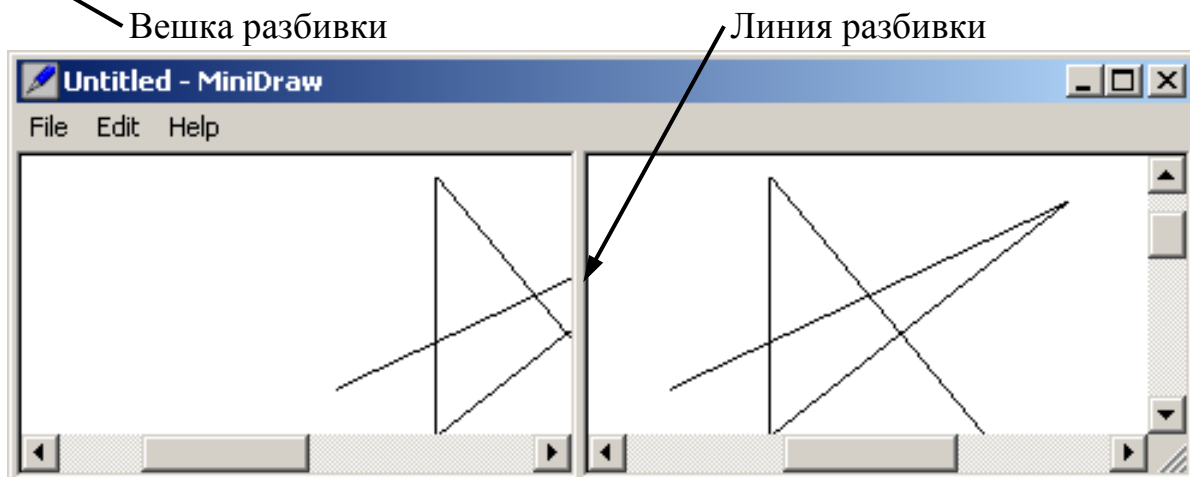
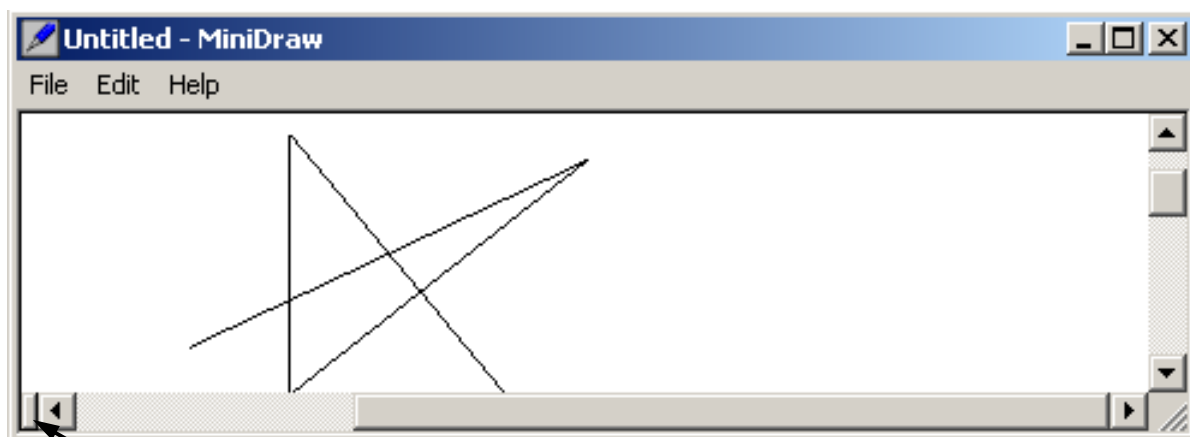
```

void CMiniDrawView::OnMouseMove(UINT nFlags, CPoint point)
{
    // TODO: Добавьте собственный код обработчика
    CClientDC ClientDC (this);
    OnPrepareDC (&ClientDC);
    ClientDC.DPtoLP (&point);
    CSize ScrollSize = GetTotalSize ();
    CRect ScrollRect (0, 0, ScrollSize.cx, ScrollSize.cy);
    if (ScrollRect.PtInRect (point))
        ::SetCursor (m_HCross);
    else
        ::SetCursor (m_HArrow);
    if (m_Dragging)
    {
        ClientDC.SetROP2 (R2_NOT);
        // ...
    }
}

```

9.2. Добавление средств разделения окна

Добавим в программу MiniDraw средства, позволяющие разделить окно представления на две части (панели), позволяющие отображать один и тот же рисунок, но прокручивать изображения (в горизонтальном направлении) независимо – см. рисунок.



Для этого к окну программы будет добавлена вешка разбивки, при двойном щелчке на которой окно разделяется на две независимые части линией разбивки. Двигая ее, можно изменить размеры панелей, а при двойном щелчке на линии разбиение снимается.

Прежде всего добавим объявление объекта `m_SplitterWnd` в определении класса `CMainFrame` (файл `MainFrm.h`):

```
class CMainFrame : public CFrameWnd
{
protected:
    CSplitterWnd m_SplitterWnd;

    // Остальная часть определения класса CMainFrame ...
}
```

Новый объект `m_SplitterWnd` – это экземпляр класса `CSplitterWnd`, служащего для создания и управления разделенным окном. Далее необходимо переопределить виртуальную функцию `OnCreateClient` как функции-члена класса `CMainFrame`. Для создания минимального определения функции вызовем `ClassWizard` (`Ctrl+W`), выберем вкладку `Message Map`, выберем класс `CMainFrame` в списках `Class name` и `Object IDs`, а затем выберем виртуальную функцию `OnCreateClient` в списке `Messages`. Добавим код в определение функции в файле `MainFrm.cpp`:

```
BOOL CMainFrame::OnCreateClient(LPCREATESTRUCT lpcs,
                                CCreateContext* pContext)
{
    // TODO: Добавьте собственный код обработчика
    return m_SplitterWnd.Create
        (this,                // родительское окно разделенного окна
         1,                    // максимальное число строк
         2,                    // максимальное число столбцов
         CSize (15, 15),       // мин. размер окна представления
         pContext);           // информация о контексте устройства
}
```

Переопределенная функция вызывает функцию `CSplitterWnd::Create`, чтобы создать разделенное окно *вместо* окна представления. Оно будет содержать только одну панель. Вторая панель будет создана при выполнении двойного щелчка на вешке разбивки.

9.3. Обновление окна представления

Каждая панель управляется отдельным окном представления (экземпляр класса `CMiniDrawView`). MFC вызывает функцию `CMiniDrawView::OnDraw` при каждом изменении данных в окне. Если в одном появилась линия, то нужно перерисовать и второе окно. Для этого

следует вызвать функцию `CDocument::UpdateAllViews` для объекта документа, чтобы MFC вызвала функцию `OnDraw` для другого представления:

```
void CMiniDrawView::OnLButtonUp(UINT nFlags, CPoint point)
{
    // ...
    PDoc->AddLine(m_PointOrigin.x, m_PointOrigin.y,
        point.x, point.y);
    PDoc->UpdateAllViews (this);
}
CScrollView::OnLButtonUp(nFlags, point);
}
```

Функция `UpdateAllViews` вынуждает программу вызвать функцию `OnDraw` для всех представлений, кроме указанного в первом аргументе (если он равен 0, обновляются все).

Эффективная перерисовка

Добьемся того, чтобы при рисовании линии в одной панели вторая перерисовывала только измененную и видимую часть рисунка. На эффективности данной программы это практически не отразится, однако в более сложных программах этот прием позволяет существенно повысить эффективность. Добавим код в файле `MiniDrawDoc.h`:

```
class CLine : public CObject
{
    // ...
    void Draw (CDC *PDC);
    CRect GetDimRect ();
    virtual void Serialize (CArchive& ar);
}
```

В конце файла `MiniDrawDoc.cpp` добавим код функции `GetDimRect`:

```
CRect CLine::GetDimRect()
{
    return CRect (min (m_X1, m_X2), min (m_Y1, m_Y2),
        max (m_X1, m_X2)+1, max (m_Y1, m_Y2)+1);
}
```

Функция `GetDimRect` возвращает объект `CRect` с размерами ограничивающего прямоугольника для линии. Макрокоманды `min` и `max` определены в `Windows.h`.

Изменим функцию `AddLine` так, чтобы она возвращала объект класса `CLine` (файл `MiniDrawDoc.h`):

```
CLine *AddLine (int X1, int Y1, int X2, int Y2);
```

В файле MiniDrawDoc.cpp произведем следующие изменения:

```
CLine *CMiniDrawDoc::AddLine (int X1, int Y1, int X2, int Y2)
{
    CLine *PLine = new CLine (X1, Y1, X2, Y2);
    m_LineArray.Add (PLine);
    SetModifiedFlag ();
    return PLine;
}
```

В файле MiniDrawView.cpp изменим код функции OnLButtonUp:

```
void CMiniDrawView::OnLButtonUp(UINT nFlags, CPoint point)
{
    // ...
    CMiniDrawDoc* PDoc = GetDocument ();
    CLine *PCLine;
    PCLine=PDoc->AddLine (m_PointOrigin.x, m_PointOrigin.y,
                          point.x, point.y);
    PDoc->UpdateAllViews(this, 0, PCLine);
    // ...
}
```

Функция UpdateAllViews класса CDocument имеет следующий формат:

```
void UpdateAllViews ( CView* pSender, LPARAM lHint = 0L,
                     CObject* pHint = NULL );
```

Необязательные второй и третий параметр передают информацию об изменениях (т.н. *рекомендацию*), которые надо внести в документ. Функция UpdateAllViews вызывает виртуальную функцию OnUpdate для каждого объекта представления, передавая ей параметры lHint и pHint. Функция OnUpdate класса CView игнорирует эти значения и перерисовывает *все окно*. Для увеличения эффективности нужно переопределить функцию OnUpdate.

Сначала для создания минимального определения функции вызовем ClassWizard (Ctrl+W), выберем вкладку Message Map, выберем класс CMiniDrawView в списках Class name и Object IDs, а затем выберем виртуальную функцию OnUpdate в списке Messages. Добавим код в определение функции в файле MiniDrawView.cpp:

```
void CMiniDrawView::OnUpdate(CView* pSender, LPARAM lHint,
                             CObject* pHint)
{
    if (pHint != 0)
    {
        CRect InvalidRect = ((CLine *)pHint)->GetDimRect ();
        CClientDC ClientDC (this);
        // Скорректировать для текущей позиции прокрутки
        OnPrepareDC (&ClientDC);
    }
}
```

```

        // Логические координаты -> физические
        ClientDC.LPtoDP (&InvalidRect);

        InvalidateRect (&InvalidRect);
    }
    else
        CScrollView::OnUpdate (pSender, lHint, pHint);
}

```

Функция OnUpdate вызывается и перед первым отображением рисунка функцией OnInitialUpdate. В этом случае pHint равно 0 (что и проверяется). Функция InvalidateRect объявляет область недействительной (требующей перерисовки), если она в данный момент отображается.

Изменим функцию OnDraw так, чтобы она перерисовывала только область, объявленную недействительной (файл MiniDrawView.cpp):

```

void CMiniDrawView::OnDraw(CDC* pDC)
{
    // ...
    pDC->LineTo (0, ScrollSize.cy);
    CRect ClipRect;
    CRect DimRect;
    CRect IntRect;
    CLine *PLine;
    pDC->GetClipBox (&ClipRect); // Получить недейств. область

    int Index = pDoc->GetNumLines ();
    while (Index--)
    {
        PLine = pDoc->GetLine (Index);
        DimRect = PLine->GetDimRect ();
        if (IntRect.IntersectRect (DimRect, ClipRect))
            PLine->Draw (pDC);
    }
}

```

Теперь можно протестировать работу приложения.

Тема 10. Перемещаемые панели и строки состояния

Перемещаемая панель инструментов (tool bar) – совокупность кнопок. Ее можно "закреплять" на краю окна или перемещать в окне.

Строка состояния (status bar) обычно отображается внизу главного окна программы и служит для вывода сообщений, отображения состояний клавиш или указания режимов работы программы.

Диалоговая панель напоминает панель инструментов, но основана на шаблоне диалогового окна и может включать кроме кнопок другие элементы управления. Будут рассмотрены в следующей теме.

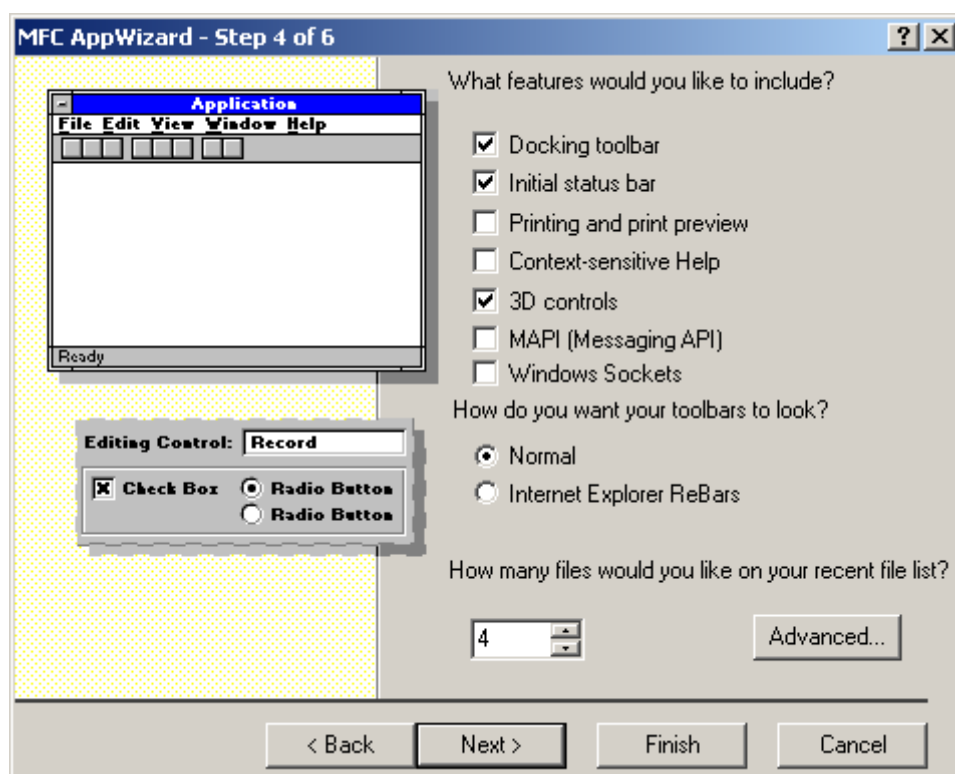
Переключаемая панель (rebar control) – контейнер для переупорядочиваемых панелей инструментов и других элементов управления. Основана на классе CReBar или CReBarControl. В качестве примера можно привести панель инструментов в Internet Explorer.

В рамках данной темы будут рассмотрены следующие вопросы:

- Добавление в новую программу перемещаемой панели инструментов и строки состояния
- Добавление перемещаемой панели инструментов в программу MiniDraw
- Добавление строки состояния в программу MiniDraw

10.1. Добавление в новую программу перемещаемой панели инструментов и строки состояния

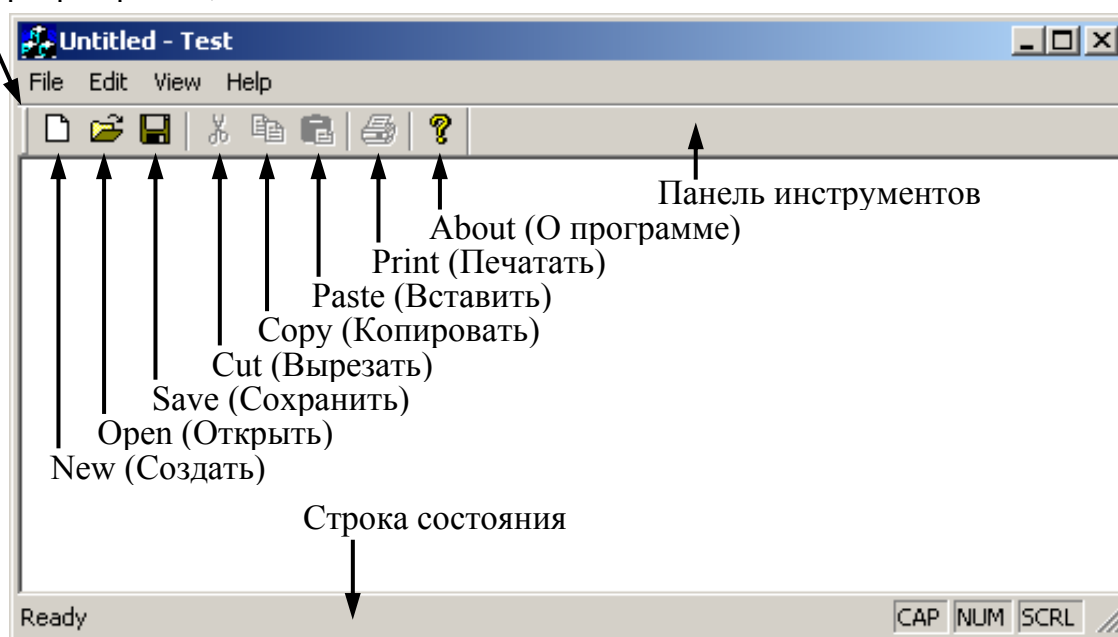
При создании нового приложения с помощью AppWizard в главное окно программы можно включить перемещаемую панель инструментов и



строку состояния. Для этого достаточно на 4 шаге (см. следующий рисунок) установить флажки Docking toolbar и Initial status bar. После запроса "How do you want your toolbars to look?" выбрать вариант Normal.

Каждая кнопка в созданной панели инструментов (см. рисунок) кроме Print соответствует команде меню, и ее нажатие приводит к выполнению тех же функций, поскольку им назначен один и тот же идентификатор ресурсов. Часть кнопок (и пунктов меню) заблокирована, поскольку им в программе не назначены обработчики.

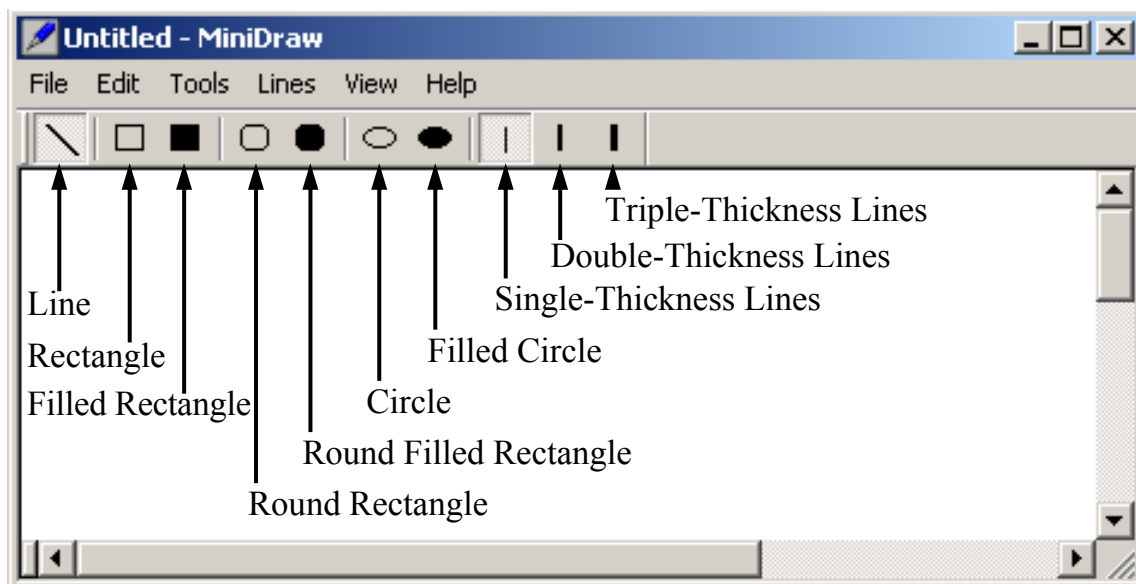
Маркер перемещения



При помещении указателя над кнопкой она приобретает трехмерный вид и появляется всплывающая подсказка, а в строке состояния появляется интерактивная справка. Их текст можно изменить, используя ClassWizard. Правая часть строки состояния называется индикатором. Текст, отображаемый в строке состояния, также можно менять программно.

10.2. Добавление перемещаемой панели инструментов в программу *MiniDraw*

Добавим в программу MiniDraw перемещаемую панель инструментов, содержащую десять кнопок. Первые семь предназначены для выбора инструментов рисования (код собственно рисования будет добавлен в программу позже), последние три — для задания ширины линии. Разрешается одновременно выбирать по одной кнопке в каждой группе. Также будут добавлены соответствующие пункты меню.



Определение ресурсов

Сначала создадим панель инструментов. На вкладке ResourceView создадим новую панель (Toolbar) – правая кнопка мыши либо Ctrl+R. В ее свойствах поменяем предложенный ID на IDR_MAINFRAME (общий идентификатор для ресурсов, связанных с главным окном).

Двойным щелчком откроем редактор панели и создадим кнопки (раздвигаются они перетаскиванием). Назначим им идентификаторы:

- ID_TOOLS_LINE
- ID_TOOLS_RECTANGLE
- ID_TOOLS_RECTFILL
- ID_TOOLS_RECTROUND
- ID_TOOLS_RECTROUNDFILL
- ID_TOOLS_CIRCLE
- ID_TOOLS_CIRCLEFILL
- ID_LINE_SINGLE
- ID_LINE_DOUBLE
- ID_LINE_TRIPLE

Замечание:

Для того, чтобы удалить кнопку, ее надо перетащить за границы рисунка панели инструментов с помощью мыши.

Добавление новых команд меню

С помощью редактора меню добавим команды меню, соответствующие кнопкам панели инструментов. Будут созданы выпадающие меню Tool и Lines, а также меню View, позволяющее скрывать или отображать панель инструментов.

Для каждой команды меню мы укажем строку интерактивной справки, которую следует ввести в поле Prompt диалогового окна Menu Item Properties. В ней до символа \n содержится интерактивная справка, а после – всплывающая подсказка.

В таблице перечислены свойства пунктов выпадающего меню Tools:

Идентификатор	Надпись	Интерактивная справка	Другие свойства
–	&Tools	–	Popup
ID_TOOLS_LINE	&Line	Select tool to draw straight lines\nLine	–
ID_TOOLS_RECTANGLE	&Rectangle	Select tool to draw open rectangles\nRectangle	–
ID_TOOLS_RECTFILL	R&ect Fill	Select tool to draw filled rectangles\nFilled Rectangle	–
ID_TOOLS_RECTROUND	Re&ct Round	Select tool to draw open rectangles with rounded corners\nRound Rectangle	–
ID_TOOLS_RECTROUND FILL	Rec&t Round Fill	Select tool to draw filled rectangles with rounded corners\nRound Filled Rectangle	–
ID_TOOLS_CIRCLE	C&ircle	Select tool to draw open circles or ellipses\nCircle	–
ID_TOOLS_CIRCLE FILL	Circle &Fill	Select tool to draw filled circles or ellipses\nFilled Circle	–
ID_TOOLS_RECTROUND	Re&ct Round	Select tool to draw open rectangles with rounded corners\nRound Rectangle	–

Аналогичная таблица для меню Lines и View:

Идентификатор	Надпись	Интерактивная справка	Другие свойства
—	&Lines	—	Popup
ID_LINE_SINGLE	&Single	Draw using single-thickness lines\nSingle-Thickness Lines	—
ID_LINE_DOUBLE	&Double	Draw using double-thickness lines\nDouble-Thickness Lines	—
ID_LINE_TRIPLE	&Triple	Draw using triple-thickness lines\nTriple-Thickness Lines	—
—	&View	—	Popup
ID_VIEW_TOOLBAR	&Toolbar	Show or hide the toolbar	—

Изменение текста программы

В файле заголовков MainFrm.h в разделе protected класса CMainFrame определим переменную m_ToolBar:

```
CToolBar m_ToolBar;
```

Для того, чтобы создать панель, необходимо добавить обработчик сообщения WM_CREATE, передаваемого в момент создания окна непосредственно перед тем, как оно станет видимым. Для создания минимального определения функции вызовем ClassWizard (Ctrl+W), выберем вкладку Message Map, выберем класс CMainFrame в списках Class name и Object IDs, а затем выберем в списке Messages идентификатор сообщения WM_CREATE. Добавим код в определение функции OnCreate в файле MainFrm.cpp.

Замечание: Если сгенерировать перемещаемую панель инструментов на этапе создания программы с использованием AppWizard, то он определит объект класса CToolBar с именем m_WndToolBar и добавит необходимые вызовы в функцию OnCreate. В коде можно будет изменить стили, передаваемые функции CToolBar::CreateEx, а также параметры, передаваемые функциям EnableDocking или убрать обращения функциям EnableDocking и DockControlBar (тогда будет создана стандартная панель инструментов).

Внесем изменения в текст функции OnCreate класса CMainFrame:

```

int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    // ...
    // TODO: Добавьте собственный код обработчика
    if (!m_ToolBar.CreateEx
        (this,
         TBSTYLE_FLAT,
         WS_CHILD | WS_VISIBLE | CBRS_TOP | CBRS_GRIPPER
         | CBRS_TOOLTIPS | CBRS_FLYBY | CBRS_SIZE_DYNAMIC))
        return -1;

    if (!m_ToolBar.LoadToolBar(IDR_MAINFRAME))
        return -1;

    m_ToolBar.EnableDocking (CBRS_ALIGN_ANY);
    EnableDocking (CBRS_ALIGN_ANY);
    DockControlBar (&m_ToolBar);

    return 0;
}

```

Функция `CToolBar::CreateEx` создает панель инструментов и задает стили. При успешном завершении возвращает 0. В противном случае главное окно удаляется и программа завершается. Значения аргументов функции:

- `this` – главное окно является родительским
- `TBSTYLE_FLAT` – панель с плоскими кнопками
- `WS_CHILD` – является дочерним окна главного окна
- `WS_VISIBLE` – делает окно видимым
- `CBRS_TOP` – задаем первоначальное размещение
- `CBRS_GRIPPER` – отображение маркера перемещения
- `CBRS_TOOLTIPS` – активизирует режим отображения всплывающих подсказок
- `CBRS_FLYBY` – отображение подсказки в строке состояния
- `CBRS_SIZE_DYNAMIC` – позволяет изменять форму панели и перемещать кнопки

Функция `CToolBar::LoadToolBar` загружает ресурс панели инструментов.

Последние три вызова позволяют панель перемещаться. Первый вызов функции `EnableDocking` (класса `CControlBar`) разрешает прикрепление панели. Второй вызов `EnableDocking` (класса `CFrameWnd`) разрешает ее перемещение в главном окне. Наконец, `CFrameWnd::DockControlBar` помещает ее на исходное место.

Написание обработчиков сообщений

Для определения и реализации обработчиков сообщений для кнопок панели инструментов и соответствующих команд меню сделаем следующее.

В окне мастера ClassWizard откроем вкладку Message Map, в списке Class name выберем класс CMiniDrawApp, так как выбор инструментов и толщины линий действует на работу всего приложения.

В списке Object IDs выберем идентификатор ID_LINE_DOUBLE, в списке Messages – сообщение COMMAND и добавим функцию (имя по умолчанию OnLineDouble). Эта функция получает управление при щелчке на кнопке, либо выборе соответствующего пункта меню.

Далее для того же идентификатора добавим функцию обработки сообщения UPDATE_COMMAND_UI (предлагаемое по умолчанию имя OnUpdateLineDouble). Эта функция получает управление через равные промежутки времени при простое системы для обновления кнопки Double-Thickness Lines, а также при открытии всплывающего меню Lines для инициализации команды меню Double.

Аналогичные действия надо сделать для идентификаторов всех кнопок панели инструментов.

Реализация обработчиков сообщений

Для реализации обработчиков сообщений сначала добавим описание двух переменных в классе CMiniDrawApp, файл MiniDraw.h:

```
class CMiniDrawApp : public CWinApp
{
public:
    int m_CurrentThickness;
    UINT m_CurrentTool;
    // ...
```

Добавим код инициализации в конструкторе класса CMiniDrawApp, файл MiniDraw.cpp:

```
CMiniDrawApp::CMiniDrawApp()
{
    // TODO: добавьте код конструктора
    m_CurrentThickness = 1;
    m_CurrentTool = ID_TOOLS_LINE;
}
```

Добавим код для вновь созданных обработчиков сообщений в файл MiniDraw.cpp:

```

////////////////////////////////////
// CMiniDrawApp commands

```

```

void CMiniDrawApp::OnLineDouble()
{
    // TODO: Добавьте собственный код обработчика
    m_CurrentThickness = 2;
}

```

```

void CMiniDrawApp::OnUpdateLineDouble(CCmdUI* pCmdUI)
{
    // TODO: Добавьте собственный код обработчика
    pCmdUI->SetCheck (m_CurrentThickness == 2 ? 1 : 0);
}

```

Функция `CCmdUI::SetCheck` с аргументом 1 делает соответствующую команду меню помеченной (если она вызвана для меню), либо кнопку панели вдавливает (если вызвана для кнопки). Код функций для других кнопок и пунктов меню, задающих толщину линии, аналогичен приведенному.

Пример кода для обработчиков сообщений, связанных с выбором инструмента рисования:

```

void CMiniDrawApp::OnToolsCircle()
{
    // TODO: Добавьте собственный код обработчика
    m_CurrentTool = ID_TOOLS_CIRCLE;
}

```

```

void CMiniDrawApp::OnUpdateToolsCircle(CCmdUI* pCmdUI)
{
    // TODO: Добавьте собственный код обработчика
    pCmdUI->SetCheck (m_CurrentTool==ID_TOOLS_CIRCLE ? 1 : 0);
}

```

Здесь приведен код только для сообщений, связанных с идентификатором `ID_TOOLS_CIRCLE`. Код функций для других идентификаторов аналогичен приведенному.

Из других функций класса `CCmdUI` для управления интерфейсом можно упомянуть следующие: `Enable`, `SetCheck`, `SetRadio`, `SetText`.

10.3. Добавление строки состояния в программу *MiniDraw*

Необходимые объявления

Чтобы добавить в MFC-программу строку состояния, необходимо определить объект класса `CStatusBar` как член главного окна, затем массив, хранящий идентификаторы полей строки состояния, после чего вызвать две функции (`Create` и `SetIndicators`) класса `CStatusBar` из функции `OnCreate` класса главного окна.

Добавим определение объекта класса `CStatusBar` как члена класса `CMainFrame` (файл `MainFrm.h`):

```
class CMainFrame : public CFrameWnd
{
protected:
    CSplitterWnd m_SplitterWnd;
    CStatusBar m_StatusBar;
    CToolBar m_ToolBar;
// ...
```

В файле `MainFrm.cpp` добавим определение массива `IndicatorIDs`:

```
// ...
BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
// ...
END_MESSAGE_MAP()
// IDs for status bar indicators:
static UINT IndicatorIDs [] = {
    ID_SEPARATOR,
    ID_INDICATOR_CAPS,
    ID_INDICATOR_NUM,
    ID_INDICATOR_SCRL,
};
// ...
```

Массив `IndicatorIDs` хранит идентификатор каждого поля, отображаемого в строке состояния. Идентификатор `ID_SEPARATOR` предназначен для создания пропуска. Он стоит первым, и строка будет иметь слева пустое поле. Остальные три нужны для отображения текущего состояния клавиш `Caps Lock`, `Num Lock` и `Scroll Lock`. Они будут выравниваться по правому краю строки состояния.

Наконец в файле `MainFrm.cpp` добавим в функцию `OnCreate` код для создания строки состояния:

```

int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
// ...
DockControlBar (&m_ToolBar);
If (!m_StatusBar.Create (this) ||
    !m_StatusBar.SetIndicators (IndicatorIDs,
    sizeof (IndicatorIDs) / sizeof (UINT)))
    return -1;
return 0;
}

```

Замечание: Здесь добавляются только стандартные поля, поддерживаемые библиотекой MFC. Чтобы добавить свое поле потребуется включить его идентификатор в массив, а также посредством редактора строк определить строковый ресурс с таким идентификатором. Кроме того, чтобы эта строка отображалась, нужно для нее создать обработчик UPDATE_COMMAND_UI и в нем вызвать функцию CCmdUI::Enable. Если нужно отображать разные строки, можно воспользоваться функцией CCmdUI::SetText.

Завершение создания меню View

В меню View добавим команду, скрывающую или отображающую строку состояния:

Идентификатор	Надпись	Интерактивная справка	Другие свойства
ID_VIEW_STATUS_BAR	&Status	Show or hide the status bar	—

Для этой команды, так же как и для команды Toolbar не требуется писать обработчик. Функция MFC CFrameWnd::OnCheckBar обрабатывает обе эти команды.

Изменение интерактивной справки

Текст интерактивной справки и всплывающей подсказки можно отредактировать, войдя в редакторе ресурсов в свойства этого элемента и изменив текст в поле Prompt, а также в редакторе строковых ресурсов. Так, например, стандартное сообщение о простое (Ready) можно изменить посредством редактирования строкового ресурса с идентификатором AFX_IDS_IDLEMESSAGE.

Тема 11. Создание диалоговых окон

Диалоговые окна очень широко используются Windows для отображения и получения данных.

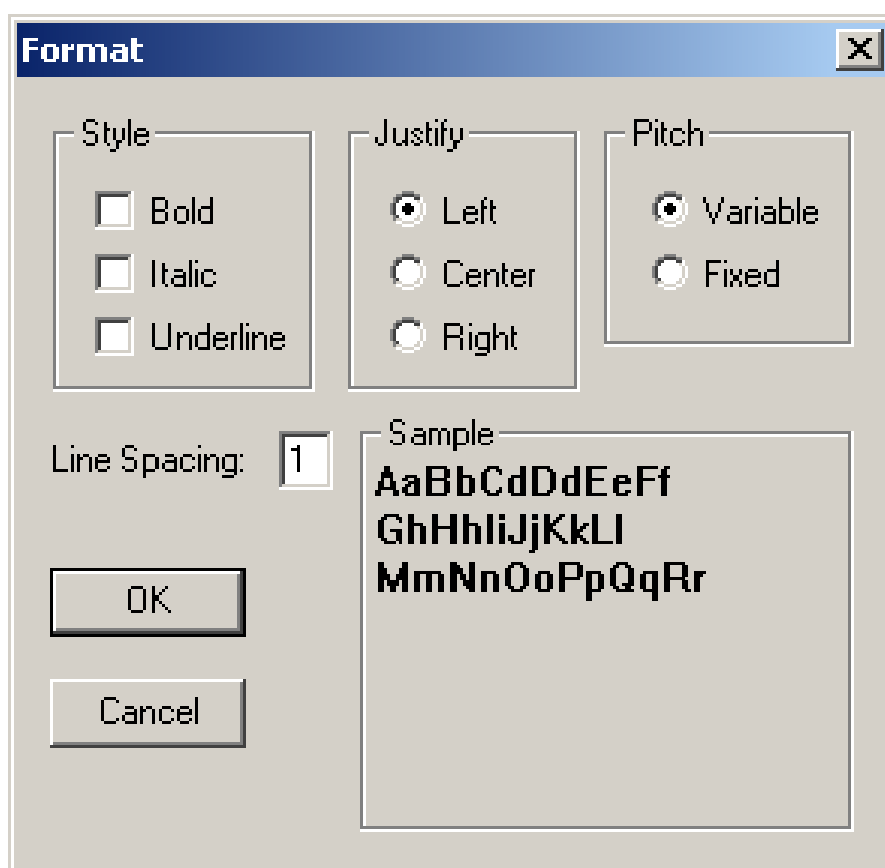
Цель изучения данной темы – научиться создавать диалоговые окна и отображать их в программах.

Будут рассмотрены следующие вопросы:

- Создание модальных диалоговых окон
- Создание немодальных диалоговых окон
- Создание диалоговых окон с вкладками
- Диалоговые окна общего назначения

11.1. Создание модальных диалоговых окон

Рассмотрим в качестве примера создание программы, отображающей несколько строк текста внутри окна представления с использованием системного шрифта Windows. По выбору команды Format в меню Text или нажатии клавиш Ctrl+A она должна выводить следующее диалоговое окно для выбора форматирования:



Выбранный стиль при нажатии кнопки OK должен применяться для отображения текста в окне.

При отображении этого диалогового окна нельзя активизировать главное окно программы или выбрать команду меню. Такие диалоговые окна называются модальными.

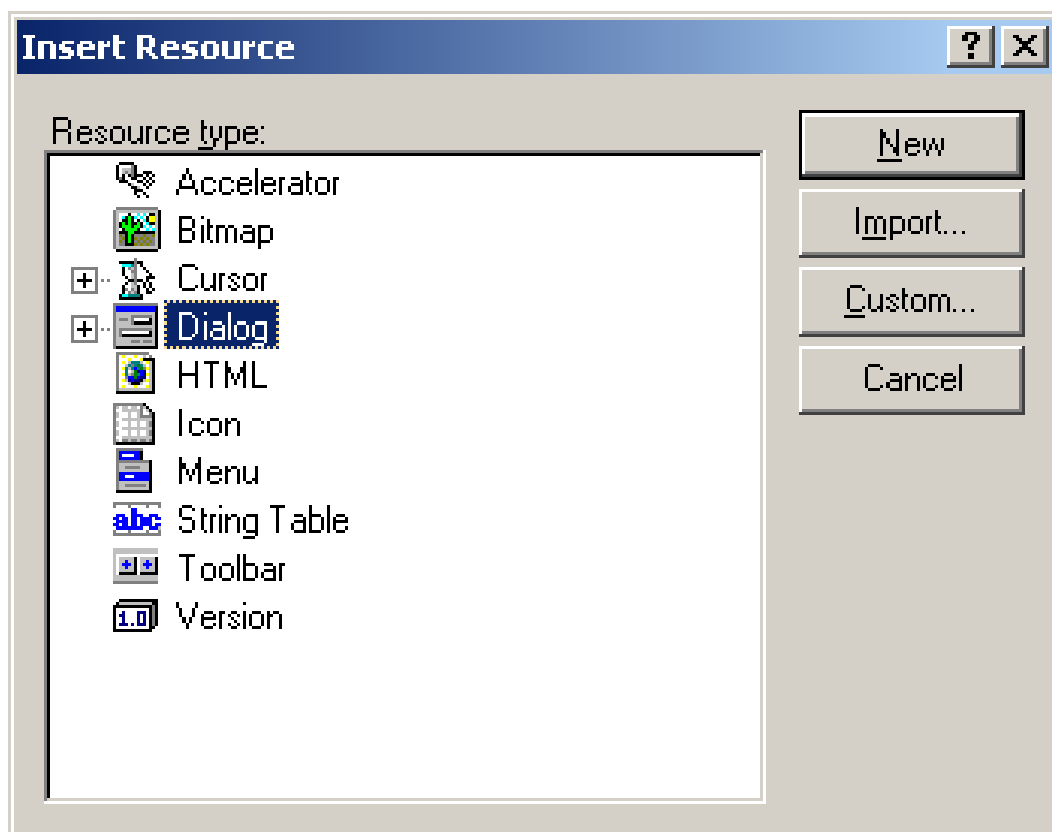
Замечание: Системный шрифт с переменной шириной не может быть полужирным.

Создание программы

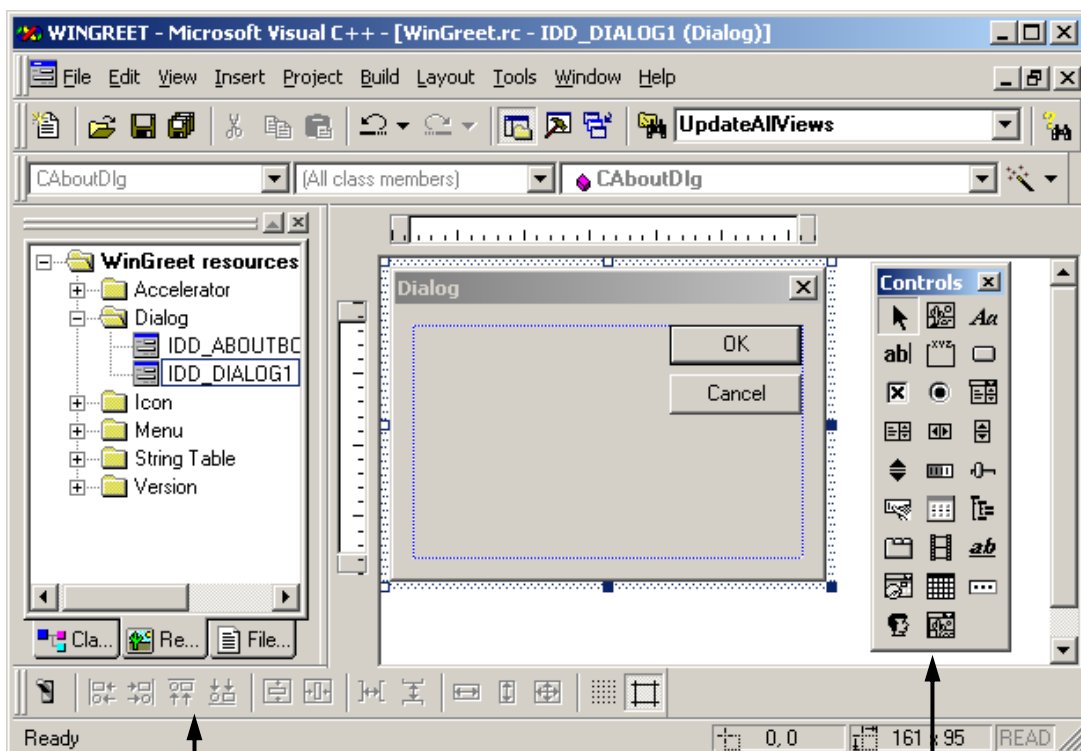
Для создания исходных файлов программы (назовем ее FontDemo) воспользуемся мастером AppWizard. В диалоговых окнах мастера выберем все те же установки (за исключением имени приложения – FontDemo), что и для программы WinGreet.

Для разработки диалогового окна Format воспользуемся редактором диалоговых окон. Выберем команду Resource в меню Insert (или Ctrl+R).

В появившемся диалоговом окне выберем тип ресурса Dialog и нажмем кнопку New.



На следующих рисунках показан вид рабочего окна редактора диалоговых окон и панели элементов управления, используемых при создании диалога.



Окно редактора диалоговых окон

Панель инструментов Dialog

Панель инструментов Controls

- | | | |
|--|--|--|
| Выбор объектов (select) | | Рисунок (picture control) |
| Надпись (static text control) | | Поле (edit box) |
| Рамка (group box) | | Кнопка (push button) |
| Флажок (check box) | | Переключатель (radio button) |
| Поле со списком (combo box) | | Список (list box) |
| Горизонтальная полоса прокрутки | | Вертикальная полоса прокрутки |
| Счетчик (spin button control) | | Индикатор (progress bar) |
| Регулятор (slider control или track bar) | | Горячая клавиша (hot-key control) |
| Окно списка (list control) | | Дерево (tree control) |
| Набор вкладок (tab control) | | Анимация (animation control) |
| Расширенное поле (rich edit box) | | Дата/время (date/time picker control) |
| Месячный календарь (month calendar) | | IP-адрес (IP-address control) |
| Пользовательский (custom control) | | Расширенное поле со списком (dropdown combo box) |

Диалоговое окно *Format* и его элементы управления

В приведенной ниже таблице перечислены свойства создаваемого диалогового окна и его элементов управления.

Идентификатор	Тип элемента управления	Надпись	Свойства, не задаваемые по умолчанию
IDD_DIALOG1	Диалоговое окно	Format	—
IDC_STATIC	Рамка	Style	—
IDC_BOLD	Флажок	&Bold	Group, Tab Stop
IDC_ITALIC	Флажок	&Italic	—
IDC_UNDERLINE	Флажок	&Underline	—
IDC_STATIC	Рамка	Justify	—
IDC_LEFT	Переключатель	&Left	Group, Tab Stop
IDC_CENTER	Переключатель	&Center	—
IDC_RIGHT	Переключатель	&Right	—
IDC_STATIC	Рамка	Pitch	—
IDC_VARIABLE	Переключатель	&Variable	Group, Tab Stop
IDC_FIXED	Переключатель	&Fixed	—
IDC_STATIC	Надпись	Line &Spacing:	—
IDC_SPACING	Поле	—	—
IDC_SAMPLE	Рамка	Sample	—
IDOK	Кнопка	OK	Default button
IDCANCEL	Кнопка	Cancel	—

Напомним, что символ & приводит к подчеркиванию следующего за ним символа и возможности выбора элемента комбинацией Alt+символ.

Задание порядка обхода элементов управления

Порядок обхода определяет последовательность получения фокуса ввода элементами с установленным свойством Tab Stop при нажатии клавиши Tab или Shift+Tab (прямое или обратное направление).

Порядок обхода используется для определения групп элементов управления. Если элементу задано свойство Group, то он и все следующие за ним (пока не встретится элемент со свойством Group) принадлежат одной группе. Если несколько переключателей (radio button) принадлежат

одной группе, то при щелчке на одной отметка снимается с ранее отмеченного (при условии, что они имеют свойство Auto). Для перемещения отметки внутри такой группы следует использовать клавиши $\leftarrow \rightarrow \uparrow \downarrow$.

Чтобы установить порядок обхода элементов, можно выбрать команду Tab Order в меню Layout или нажать Ctrl+D. После этого редактор диалоговых окон размещает номера на элементах управления, соответствующие порядку их создания. Чтобы изменить порядок, нужно последовательно щелкнуть на каждом элементе в нужном порядке.

На этом разработка диалогового окна завершена.

Создание класса для управления диалоговым окном

Вызовем ClassWizard (Ctrl+W). Так как мы еще не создали класс диалогового окна Format, он сразу отобразит диалоговое окно Adding A Class. Выберем в нем опцию Create A New Class и нажмем OK. Согласимся взять идентификатор IDD_DIALOG1 и базовый класс CDialog. В качестве имени класса введем CFormat, и ClassWizard автоматически предложит имя файла реализации Format.cpp (соответственно, определение в файле Format.h). Если вы хотите изменить имя файла, нажмите Change.

Определение переменных-членов класса

Вернемся в главное окно ClassWizard, откроем вкладку Member Variables. Выберем в списке Class name класс CFormat. В списке Control IDs перечислены идентификаторы элементов управления, которым можно сопоставить переменные. Выберем IDC_BOLD и щелкнем кнопку Add Variable. В окне Add Member Variable введем имя m_Bold, категорию Value и тип BOOL, затем нажмем OK. Аналогичным образом введем переменные m_Italic и m_Underline.

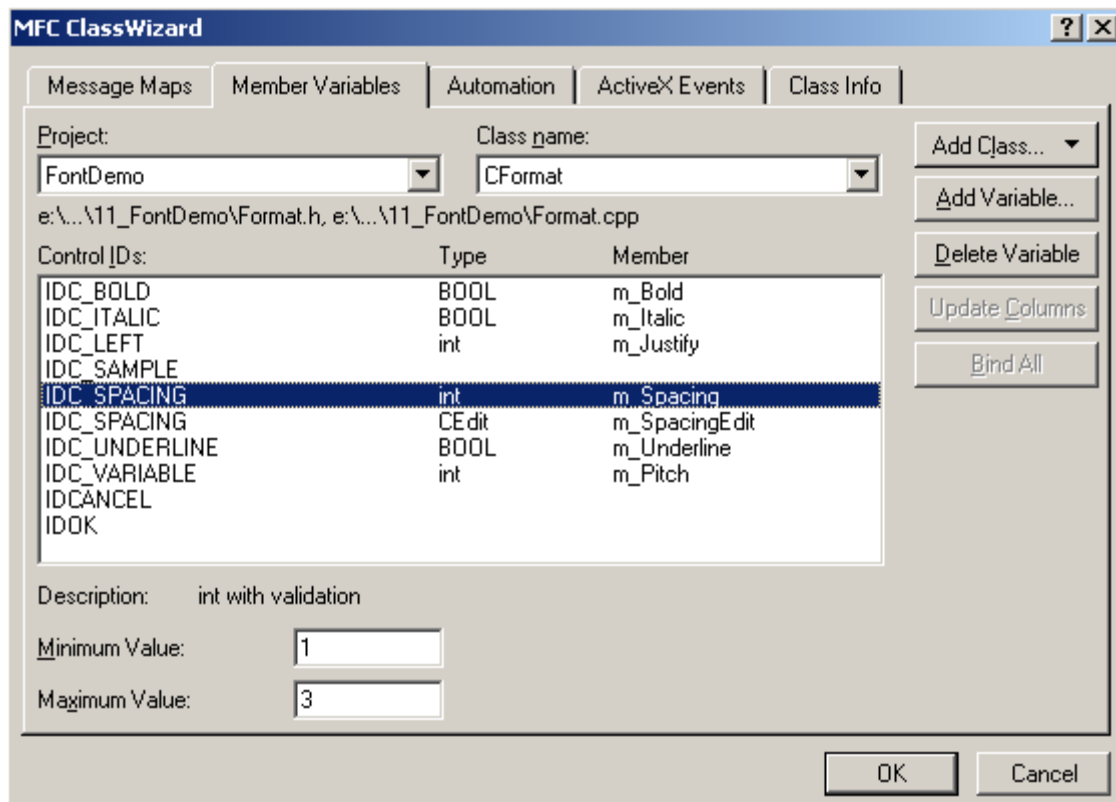
При задании переменных для переключателей следует иметь в виду, что переменная назначается на группу переключателей, а не на каждый из них. Она содержит номер отмеченного переключателя в порядке их обхода (нумерация с нуля). Когда диалоговое окно открывается в первый раз, этой переменной присваивается -1 (ни один переключатель не отмечен). При выборе OK MFC записывает в эту переменную номер выбранного переключателя (или -1).

В списке Control IDs показаны идентификаторы первых переключателей для каждой группы: у нас это IDC_LEFT и IDC_VARIABLE. Введем для них переменные типа int с именами m_Justify и m_Pitch соответственно.

Введем переменную m_Spacing для поля (IDC_SPACING). В качестве типа выберем int вместо предлагаемого по умолчанию CString для обеспечения преобразования этой строки в целое и обратно. После возвращения в ClassWizard в окне будут отображены поля Minimum Value и Maximum Value, предназначенные для задания ограничений на значение этой переменной. Зададим значения 1 и 3.

Наконец, определим переменную для управления полем. Выберем IDC_SPACING и добавим переменную m_SpacingEdit, причем в списке Category выберем Control. Эта переменная теперь имеет тип CEdit, этот класс предоставляет набор функций для управления полем.

Окно MFC ClassWizard должно приобрести следующий вид.



Нажмем кнопку ОК. Мастер запишет код определения переменных в файл заголовков и файл реализации класса диалогового окна – у нас Format.h и Format.cpp соответственно.

Определение обработчиков событий

Теперь необходимо написать обработчики событий, получаемых диалоговым окном Format, а именно событий, посылаемых при:

- первом открытии диалогового окна,
- его перерисовке,
- щелчке на флажке или переключателе.

В окне ClassWizard откроем вкладку Message Map. Выберем класс CFormat. Для создания обработчиков первых двух событий в окне Object IDs выберем CFormat. В списке Messages отобразятся идентификаторы уведомляющих сообщений Windows, посылаемых диалоговому окну. Выберем сообщение WM_INITDIALOG и добавим функцию (по умолчанию имя OnInitDialog). Затем для сообщения WM_PAINT также создадим обработчик (OnPaint).

Теперь создадим обработчики сообщений для каждого флажка и переключателя. выберем в списке Object IDs идентификатор IDC_BOLD. В окне отобразятся два сообщения: BN_CLICKED и BN_DOUBLECLICKED. Выберем первое. Добавим функцию, согласившись с предложенным по умолчанию именем. Прделаем эту процедуру для всех флажков и переключателей.

Для определения функции, которая получает управление при изменении содержимого поля Spacing, выберем в Object IDs IDC_SPACING, а в списке Messages сообщение EN_CHANGE и добавим функцию (имя по умолчанию).

Управление диалоговым окном класса CFormat

В файле Format.h добавим два определения в начало файла:

```
enum {JUSTIFY_LEFT, JUSTIFY_CENTER, JUSTIFY_RIGHT};
enum {PITCH_VARIABLE, PITCH_FIXED};
```

```
////////////////////////////////////
// CFormat dialog
```

Здесь перечисляемые константы используются для обращения к переключателям группы Justify и Pitch соответственно. Добавим определение m_RectSample – переменной-члена класса CFormat:

```
////////////////////////////////////
// CFormat dialog
```

```
class CFormat : public CDialog
{
protected:
    RECT m_RectSample;
    // ...
```

В файле Format.cpp добавим код в функцию OnInitDialog:

```
BOOL CFormat::OnInitDialog()
{
    CDialog::OnInitDialog();
    // TODO: добавьте код инициализации
    GetDlgItem (IDC_SAMPLE)->GetWindowRect (&m_RectSample);
    ScreenToClient (&m_RectSample);
    m_SpacingEdit.LimitText (1);
    return TRUE;
}
```

Функция OnInitDialog получает управление при открытии диалогового окна непосредственно перед его отображением. Первый добавленный опе-

ратор сохраняет экранные координаты рамки Sample в переменной m_RectSample. Второй преобразует их в координаты окна. Третий ограничивает количество вводимых символов поле ввода одним, вызывая функцию LimitText класса CEdit.

Замечание: Функция OnInitDialog возвращает TRUE, если фокус не установлен на элементе управления.

MFC-классы для элементов управления

MFC-классы для манипулирования элементами управления в большинстве своем порождены от класса CWnd (некоторые – от других классов элементов управления). Это позволяет использовать члены базовых классов для получения информации или манипулирования элементами управления.

В приведенном выше коде для работы с полем ввода Spacing мы ввели переменную категории Control как члена класса CFormat. Для этих целей необязательно создавать постоянный объект. Альтернатива – использование функции CWnd::GetDlgItem:

```
((CEdit *)GetDlgItem(IDC_SPACING)) ->LimitText (1);
```

Отметим только, что функция создает *указатель на временный объект*, который действует только во время обработки текущего события и в дальнейшем *не сохраняется*.

Теперь, например, вы можете управлять переключателем Bold, если хотите, чтобы при выбранном переключателе Variable опция Bold была недоступна (мы отмечали, что шрифт System с переменной шириной не может быть полужирным). При этом для того, чтобы установить или снять отметку переключателя, можно использовать функцию CButton::SetCheck, а для разрешения или блокировки доступа к переключателю CWnd::EnableWindow.

Управление окном класса CFormat – Style

Теперь завершим программирование обработчиков сообщений. В файле Format.cpp добавим код в функцию OnBold:

```
void CFormat::OnBold()
{
    // TODO: Добавьте собственный код обработчика
    m_Bold = !m_Bold;
    InvalidateRect (&m_RectSample);
    UpdateWindow ();
}
```

Первый оператор изменяет значение переменой m_Bold на противоположное. Второй объявляет недействительной часть диалогового окна,

занятую рамкой Sample, третий приводит к непосредственному вызову функции OnPaint класса диалогового окна.

Аналогичный код добавим в функции OnItalic и OnUnderline.

Управление окном класса CFormat – Justify и Pitch

Обработчики сообщений для переключателей группы Justify:

```
void CFormat::OnLeft()
{
    if (IsDlgButtonChecked (IDC_LEFT))
    {
        m_Justify = JUSTIFY_LEFT;
        InvalidateRect (&m_RectSample);
        UpdateWindow ();
    }
}
```

Аналогичный код имеют функции OnCenter и OnRight.

Обработчики сообщений для переключателей группы Pitch:

```
void CFormat::OnFixed()
{
    if (IsDlgButtonChecked (IDC_FIXED))
    {
        m_Pitch = PITCH_FIXED;
        InvalidateRect (&m_RectSample);
        UpdateWindow ();
    }
}
```

Аналогичный код имеет функция OnVariable.

Управление окном класса CFormat – OnChangeSpacing

Добавим код в обработчике уведомления EN_CHANGE – функции OnChangeSpacing:

```
void CFormat::OnChangeSpacing()
{
    // TODO: Добавьте собственный код обработчика
    int Temp;
    Temp = (int)GetDlgItemInt (IDC_SPACING);
    if (Temp > 0 && Temp < 4)
    {
        m_Spacing = Temp;
        InvalidateRect (&m_RectSample);
        UpdateWindow ();
    }
}
```

Вызов функции `GetDlgItemInt` позволяет получить содержимое поля в виде целочисленного значения. Далее мы сохраняем это значение и перерисовываем образец текста.

Управление окном класса `CFormat` – `OnPaint`

Наконец, требуется написать код функции `OnPaint`, которая вызывается, когда требуется перерисовать окно.

```
void CFormat::OnPaint()
{
    CPaintDC dc(this); // контекст устройства для рисования
    // TODO: Добавьте собственный код обработчика
    CFont Font;
    LOGFONT LF;
    int LineHeight;
    CFont *PtrOldFont;
    int X, Y;

    // заполнение структуры LF свойствами
    // стандартного системного шрифта:
    CFont TempFont;
    if (m_Pitch == PITCH_VARIABLE)
        TempFont.CreateStockObject (SYSTEM_FONT);
    else
        TempFont.CreateStockObject (SYSTEM_FIXED_FONT);
    TempFont.GetObject (sizeof (LOGFONT), &LF);
    // инициализируем поля lfWeight, lfItalic и lfUnderline:
    if (m_Bold)
        LF.lfWeight = FW_BOLD;
    if (m_Italic)
        LF.lfItalic = 1;
    if (m_Underline)
        LF.lfUnderline = 1;

    // создание и выбор шрифта:
    Font.CreateFontIndirect (&LF);
    PtrOldFont = dc.SelectObject (&Font);
    // задаем выравнивание:
    switch (m_Justify)
    {
        case JUSTIFY_LEFT:
            dc.SetTextAlign (TA_LEFT);
            X = m_RectSample.left + 5;
            break;
```

```

    case JUSTIFY_CENTER:
        dc.SetTextAlign (TA_CENTER);
        X = (m_RectSample.left + m_RectSample.right) / 2;
        break;
    case JUSTIFY_RIGHT:
        dc.SetTextAlign (TA_RIGHT);
        X = m_RectSample.right - 5;
        break;
}
// установка режима отображения фона:
dc.SetBkMode (TRANSPARENT); // прозрачный

// вывод строк текста:
LineHeight = LF.lfHeight * m_Spacing;
Y = m_RectSample.top + 15;
dc.TextOut (X, Y, "AaBbCdDdEeFf");
Y += LineHeight;
dc.TextOut (X, Y, "GhHhIiJjKkLl");
Y += LineHeight;
dc.TextOut (X, Y, "MmNnOoPpQqRr");

// отмена выбора шрифта:
dc.SelectObject (PtrOldFont);

// Не вызывайте CDialog::OnPaint()
// для сообщений, связанных с перерисовкой
}

```

Отображение диалогового окна

Прежде всего изменим созданное мастером AppWizard меню:

1. Удалим меню File и меню Edit.
2. Добавим меню Text слева от Help.
3. Добавим к новому меню команду Format, разделитель и команду Exit.

Выпадающее меню Text:

Идентификатор	Надпись	Другие свойства
—	&Text	Popup
ID_TEXT_FORMAT	&Format...\tCtrl+F	—
—	—	Separator
ID_APP_EXIT	E&xit	—

Затем в таблицу горячих клавиш IDR_MAINFRAME добавим комбинацию клавиш Ctrl+F для команды Format (идентификатор ID_TEXT_FORMAT).

Используя мастера ClassWizard, добавим обработчика сообщений Format в класс CFontDemoDoc (идентификатор ID_TEXT_FORMAT, сообщение COMMAND, имя функции по умолчанию OnTextFormat).

В определение класса CFontDemoDoc (файл FontDemoDoc.h) добавим переменные, предназначенные для сохранения значений параметров форматирования:

```
class CFontDemoDoc : public CDocument
{
public:
    BOOL m_Bold;
    BOOL m_Italic;
    int m_Justify;
    int m_Pitch;
    int m_Spacing;
    BOOL m_Underline;
    // ...
}
```

В файле реализации класса CFontDemoDoc (FontDemoDoc.cpp) в конструкторе добавим код для инициализации только что введенных переменных:

```
CFontDemoDoc::CFontDemoDoc()
{
    // TODO: добавьте код конструктора
    m_Bold = FALSE;
    m_Italic = FALSE;
    m_Justify = JUSTIFY_LEFT;
    m_Pitch = PITCH_VARIABLE;
    m_Spacing = 1;
    m_Underline = FALSE;
}
```

В FontDemoDoc.cpp также добавим директиву для подключения файла Format.h, чтобы объявление класса диалогового окна было доступно в файле FontDemoDoc.cpp:

```
#include "FontDemoDoc.h"
#include "format.h"
```

Отображение диалогового окна – OnTextFormat

Теперь необходимо добавить код обработки в функцию OnTextFormat, сгенерированную мастером ClassWizard:

```

void CFontDemoDoc::OnTextFormat()
{
    // TODO: Добавьте собственный код обработчика

    // объявление объекта класса диалогового окна:
    CFormat FormatDlg;

    // инициализация переменных класса:
    FormatDlg.m_Bold = m_Bold;
    FormatDlg.m_Italic = m_Italic;
    FormatDlg.m_Justify = m_Justify;
    FormatDlg.m_Pitch = m_Pitch;
    FormatDlg.m_Spacing = m_Spacing;
    FormatDlg.m_Underline = m_Underline;
    // отображение диалогового окна:
    if (FormatDlg.DoModal () == IDOK)
    {
        // сохранение установленных значений:
        m_Bold = FormatDlg.m_Bold;
        m_Italic = FormatDlg.m_Italic;
        m_Justify = FormatDlg.m_Justify;
        m_Pitch = FormatDlg.m_Pitch;
        m_Spacing = FormatDlg.m_Spacing;
        m_Underline = FormatDlg.m_Underline;

        // перерисовка текста:
        UpdateAllViews (NULL);
    }
}

```

Здесь создается объект класса CFormat, и для отображения его окна используется функция DoModal, которая возвращает управление только по закрытии окна – в нашем случае она может вернуть значение IDOK или IDCANCEL.

Отображение окна класса CFontDemoView – OnDraw

Наконец, требуется написать код функции OnDraw, для класса CFontDemoView. Этот код очень похож на код функции OnPaint класса CFormat:

```

void CFontDemoView::OnDraw(CDC* pDC)
{
    CFontDemoDoc* pDoc = GetDocument()
    ASSERT_VALID(pDoc);
    // TODO: добавьте код отображения собственных данных
    RECT ClientRect;
    CFont Font;

```

```

LOGFONT LF;
int LineHeight;
CFont *PtrOldFont;
int X, Y;
// заполнение структуры LF свойствами
// стандартного системного шрифта:
CFont TempFont;
if (pDoc->m_Pitch == PITCH_VARIABLE)
    TempFont.CreateStockObject (SYSTEM_FONT);
else
    TempFont.CreateStockObject (SYSTEM_FIXED_FONT);
TempFont.GetObject (sizeof (LOGFONT), &LF);
// инициализируем поля lfWeight, lfItalic и lfUnderline:
if (pDoc->m_Bold)
    LF.lfWeight = FW_BOLD;
if (pDoc->m_Italic)
    LF.lfItalic = 1;
if (pDoc->m_Underline)
    LF.lfUnderline = 1;

// создание и выбор шрифта:
Font.CreateFontIndirect (&LF);
PtrOldFont = pDC->SelectObject (&Font);
// задаем выравнивание:
GetClientRect (&ClientRect);
switch (pDoc->m_Justify)
{
    case JUSTIFY_LEFT:
        pDC->SetTextAlign (TA_LEFT);
        X = ClientRect.left + 5;
        break;
    case JUSTIFY_CENTER:
        pDC->SetTextAlign (TA_CENTER);
        X = (ClientRect.left + ClientRect.right) / 2;
        break;
    case JUSTIFY_RIGHT:
        pDC->SetTextAlign (TA_RIGHT);
        X = ClientRect.right - 5;
        break;
}
// установка цвета текста и режима фона:
pDC->SetTextColor (::GetSysColor (COLOR_WINDOWTEXT));
pDC->SetBkMode (TRANSPARENT);

```



```

// вывод строк текста:
LineHeight = LF.lfHeight * pDoc->m_Spacing;
Y = 5;
pDC->TextOut (X, Y,
               "This is the first line of sample text.");
Y += LineHeight;
pDC->TextOut (X, Y,
               "This is the second line of sample text.");
Y += LineHeight;
pDC->TextOut (X, Y,
               "This is the third line of sample text.");
// отмена выбора шрифта:
pDC->SelectObject (PtrOldFont);
}

```

В отличие от функции OnPaint класса CFormat вместо простого вывода текста с использованием стандартного черного цвета здесь мы вызываем функции ::GetSysColor и CDC::SetBkMode, которые устанавливают цвет текста Window Font, выбранный в панели управления Windows. Цвет фона окна устанавливается MFC автоматически исходя из настроек Windows.

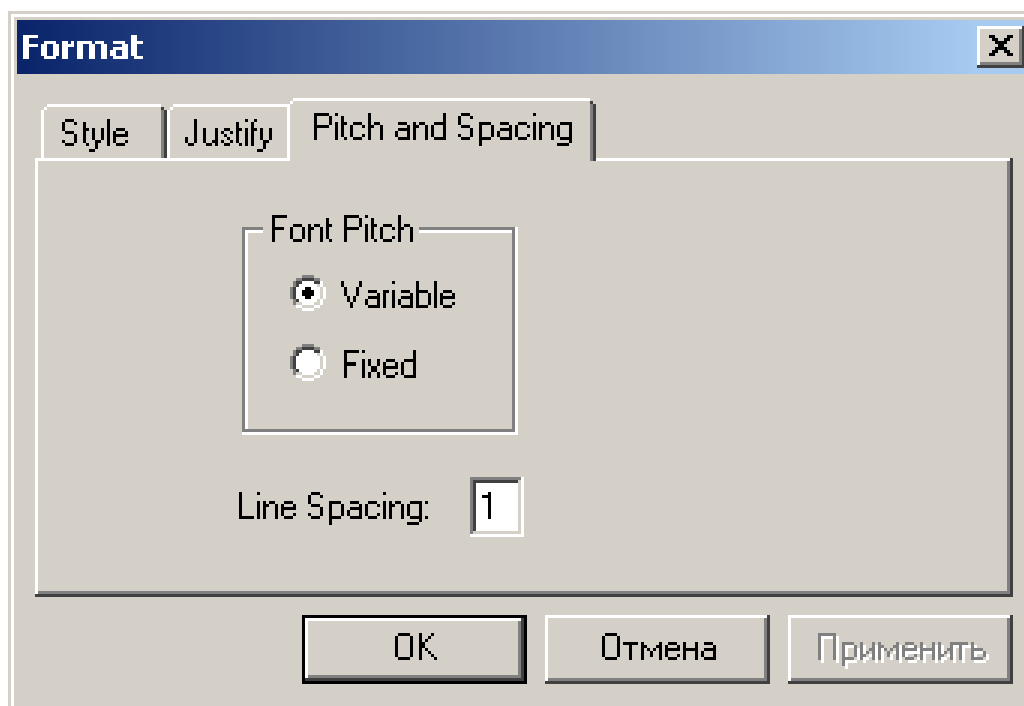
11.2. Создание немодальных диалоговых окон

При отображении модального диалогового окна главное окно программы блокируется. Для продолжения работы его необходимо закрыть. Если требуется продолжать работу программы, не закрывая диалогового окна, нужно его сделать *немодальным*. Собственно проектирование немодального окна не отличается от выше описанного. Отличие состоит в способе его отображения:

- Экземпляр класса диалогового окна следует объявить как глобальный объект или создать с помощью оператора new.
- Немодальное диалоговое окно отображается путем вызова функции CDialog::Create вместо CDialog::DoModal. Функция Create возвращает управление, оставляя диалоговое окно на экране
- Немодальное диалоговое окно закрывается путем вызова функции CDialog::DestroyWindow вместо EndDialog. Ее можно вызвать из любой функции-члена класса окна или другой функции программы.
- Необходимо определить обработчик сообщения OnCancel для класса диалогового окна. Если окно содержит кнопку с идентификатором IDOK, необходимо определить обработчик OnOK. Названные функции должны вызывать функцию DestroyWindow, закрывающую диалоговое окно и не должны вызывать обработчик сообщения базового класса. Для проверки или сохранения содержимого элементов управления функция OnOK должна вызвать функцию CWnd::UpdateData с параметром TRUE или без параметров.

11.3. Создание диалоговых окон с вкладками

Продemonстрируем процесс создания диалогового окна с вкладками на примере программы TabDemo, которая функционально равноценна программе FontDemo, но диалоговое окно Format будет выглядеть, как окно с вкладками. Диалоговое окно с вкладками поддерживается объектом класса CPropertySheet, а каждая страница – класса CPropertyPage.



CPropertySheet создает страницы по крайней мере такой же ширины, как ширина трех кнопок внизу, т.е. больше, чем необходимо для размещения необходимых элементов управления. Однако в реальных программах окна с вкладками используются, только если необходимо отобразить именно много элементов управления.

Окончательный вид создаваемого нами диалогового окна представлен на приведенном выше рисунке.

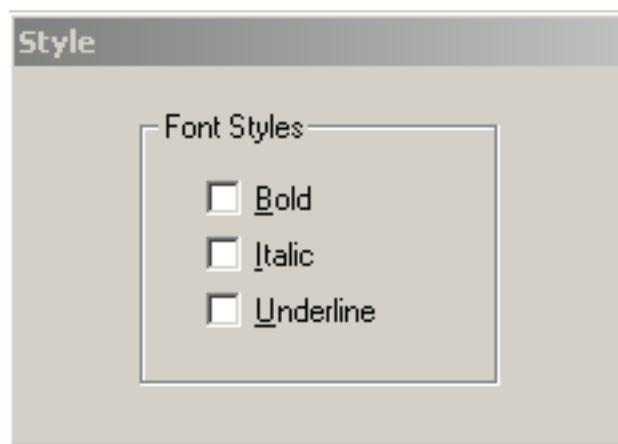
Создание шаблона диалогового окна

Создадим, как и ранее, исходные файлы программы TabDemo с помощью мастера AppWizard. Для создания шаблона диалогового окна:

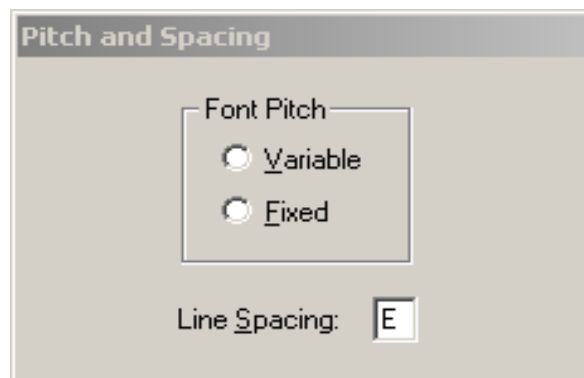
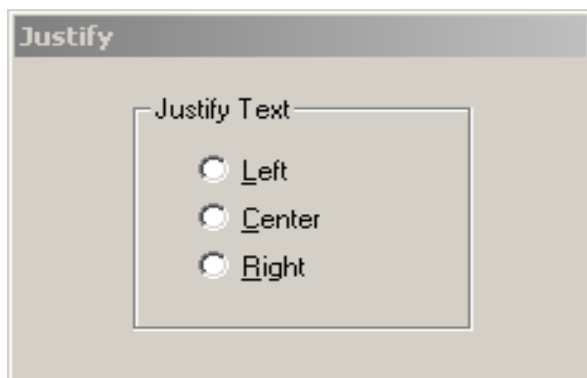
1. Создадим новый ресурс типа Dialog.
2. Откроем диалоговое окно Dialog Properties. На вкладке General в поле Caption введем Style. На вкладке Styles в списке Style выберем пункт Child, а в списке Borders – пункт Thin. Из флажков должен быть отмечен только флажок Titlebar. На вкладке More styles оставим отмеченным только флажок Disabled.
3. Удалим все кнопки и добавим элементы управления, как показано на рисунке. Свойства элементов приведены в таблице.

Идентификатор	Тип элемента управления	Надпись	Устанавливаемые свойства
IDC_STATIC	Рамка	Font Styles	—
IDC_BOLD	Флажок	&Bold	Group, Tab Stop
IDC_ITALIC	Флажок	&Italic	—
IDC_UNDERLINE	Флажок	&Underline	—
IDC_STATIC	Рамка	Justify Text	—
IDC_LEFT	Переключатель	&Left	Group, Tab Stop
IDC_CENTER	Переключатель	&Center	—
IDC_RIGHT	Переключатель	&Right	—
IDC_STATIC	Рамка	Font Pitch	—
IDC_VARIABLE	Переключатель	&Variable	Group, Tab Stop
IDC_FIXED	Переключатель	&Fixed	—
IDC_STATIC	Надпись	Line &Spacing:	—
IDC_SPACING	Поле	—	—

- Пока отображается диалоговое окно Style, запустим ClassWizard для создания класса, управляющего страницей Style. В окне Create A New Class введем имя класса CStyle, а в качестве базового класса выберем CPropertyPage.



- Как и ранее, на вкладке Member Variables определим переменные для флажков Bold, Italic и Underline.
- Аналогичным образом создадим шаблоны второго и третьего диалоговых окон для двух оставшихся страниц, по образцу, приведенному на рисунке. Имена классов должны быть CJustify и CPitch соответственно. Добавим в классы такие же переменные-члены, как и в программе FontDemo.



7. Создадим обработчика сообщения WM_INITDIALOG класса CPitch с именем OnInitDialog. Добавим в него код:

```
BOOL CPitch::OnInitDialog()
{
    // ...
    m_SpacingEdit.LimitText (1);
    return TRUE;
}
```

8. Модифицируем меню программы, удалив меню File и Edit и добавив меню Text с командами Format и Exit, а также комбинацию клавиш Ctrl+F, как это было сделано в программе FontDemo.

Сгенерируем обработчик командного сообщения для команды Format и добавим следующий код в функцию OnTextFormat в файле TabDemoDoc.cpp:

```
void CTabDemoDoc::OnTextFormat()
{
    // TODO: Добавьте собственный код обработчика
    // создание объекта диалогового окна с вкладками:
    CPropertySheet PropertySheet ("Format");

    // создание объекта для каждой страницы:
    CStyle StylePage;
    CJustify JustifyPage;
    CPitch PitchPage;

    // добавление страниц к объекту диалогового окна:
    PropertySheet.AddPage (&StylePage);
    PropertySheet.AddPage (&JustifyPage);
    PropertySheet.AddPage (&PitchPage);
    // инициализация объектов страниц:
    StylePage.m_Bold = m_Bold;
    StylePage.m_Italic = m_Italic;
    StylePage.m_Underline = m_Underline;
    JustifyPage.m_Justify = m_Justify;
```

```

PitchPage.m_Pitch = m_Pitch;
PitchPage.m_Spacing = m_Spacing;

// отображение диалогового окна с вкладками:
if (PropertySheet.DoModal () == IDOK)
{
    // сохранение значений элементов управления страниц:
    m_Bold = StylePage.m_Bold;
    m_Italic = StylePage.m_Italic;
    m_Underline = StylePage.m_Underline;
    m_Justify = JustifyPage.m_Justify;
    m_Pitch = PitchPage.m_Pitch;
    m_Spacing = PitchPage.m_Spacing;

    // перерисовка текста:
    UpdateAllViews (NULL);
}
}

```

Как и класс CDialog, класс CPropertyPage обеспечивает обработку для кнопок ОК и Cancel, также проверяет и передает данные между элементами управления и функциями объектов страниц. Чтобы сделать доступной кнопку Apply (Применить) или добавить собственную кнопку и обработчик для нее, необходимо породить собственный класс от класса CPropertyPage.

Включим в файл TabDemoDoc.cpp файлы заголовков для каждого из классов страниц, а также код инициализации необходимых переменных:

```

#include "TabDemoDoc.h"
#include "style.h"
#include "justify.h"
#include "pitch.h"

// ...
CTabDemoDoc::CTabDemoDoc ()
{
    // TODO: добавьте код конструктора
    m_Bold = FALSE;
    m_Italic = FALSE;
    m_Underline = FALSE;
    m_Justify = JUSTIFY_LEFT;
    m_Pitch = PITCH_VARIABLE;
    m_Spacing = 1;
}

```

9. В файле TabDemoDoc.h определим следующие перечисления и переменные:

```
enum {JUSTIFY_LEFT, JUSTIFY_CENTER, JUSTIFY_RIGHT};
enum {PITCH_VARIABLE, PITCH_FIXED};
class CTabDemoDoc : public CDocument
{
public:
    BOOL m_Bold;
    BOOL m_Italic;
    BOOL m_Underline;
    int m_Justify;
    int m_Pitch;
    int m_Spacing;
// ...

```

10. В файле TabDemoView.cpp добавим в функцию OnDraw такой же код, как и в программе FontDemo.

В файле TabDemo.cpp добавим в функцию InitInstance вызов функции SetWindowText:

```
BOOL CTabDemoApp::InitInstance()
{
// ...
    m_pMainWnd->SetWindowText ("Tabbed Dialog Box Demo");
    return TRUE;
}

```

Теперь можно протестировать работу приложения.

11.4. Диалоговые окна общего назначения

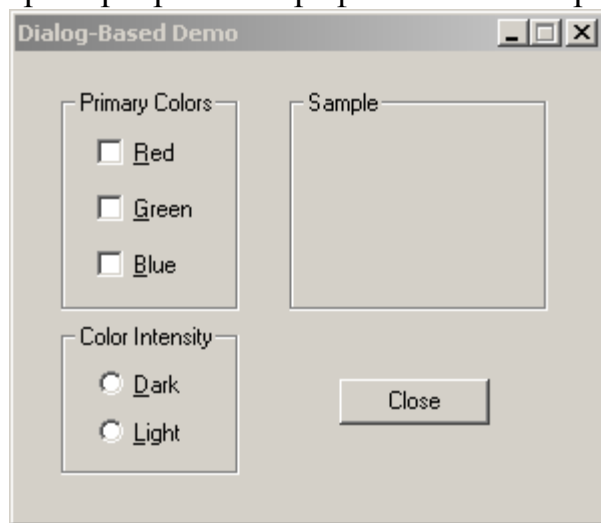
Windows предоставляет *общие диалоговые окна*, предназначенные для выполнения специфических задач, например, открытия файла или выбора цвета, а библиотека MFC – классы для управления диалоговыми окнами общего назначения всех типов:

MFC-класс	Управляемые диалоговые окна
CColorDialog	Color – выбор цвета
CFileDialog	Open – открытие файла Save As – сохранение файла под указанным именем
CFindReplaceDialog	Find – поиск текста Replace – замена текста
CFontDialog	Font – выбор шрифта текста
COleDialog	Предназначен для создания диалоговых окон приложений OLE
CPageSetupDialog	OLE Page Setup – определение установок страниц и полей печати для приложений OLE

Тема 12. Разработка диалоговых приложений

Рассмотренные нами ранее примеры программ с графическим интерфейсом содержали свободную область для отображения текста или графики. Такие программы предназначены для просмотра и редактирования документов.

В данной теме мы рассмотрим вопрос о разработке программ, не обрабатывающих документы. Это диалоговые приложения – программы, у которых в главном окне отображается только совокупность элементов управления. Примеры такого рода программ: программы поиска данных, поиска файлов, набора телефонных номеров, калькуляторы и т.п.



Мы рассмотрим процесс создания таких программ двух типов:

- Программы, отображающие диалоговое окно без главного окна или окна представления.
- Программы просмотра форм, отображающие главное окно с объектами пользовательского интерфейса и окно представления с элементами управления, основанное на шаблоне диалогового окна.

12.1. Простые диалоговые программы

Генерация исходных файлов программы DlgDemo

Выбранная в качестве примера программа DlgDemo должна позволить пользователю выбрать цвета, а затем отобразить результат их смешения в заданной области диалогового окна. Для начальной генерации исходных модулей программы воспользуемся, как всегда, мастером AppWizard.

- В диалоговом окне мастера Step 1 выберем установку Dialog based.
- Во втором диалоговом окне (Step 2) отменим выбор опций About Box, ActiveX Controls (опция 3D Controls должна остаться) и введем строку "Dialog-Based Demo" в текстовое поле Please enter a title for your dialog box.
- В следующем диалоговом окне (Step 3) выберем статическую компоновку библиотеки MFC
- В диалоговом окне Step 4 оставим все без изменений.

Мастер AppWizard сгенерирует два основных класса: приложения и диалогового окна программы. Первый имеет имя CDlgDemoApp и определен в файлах DlgDemo.h и DlgDemo.cpp. Класс диалогового окна управля-

ет окном программы. Его имя CDlgDemoDlg, он определен в файлах DlgDemoDlg.h и DlgDemoDlg.cpp.

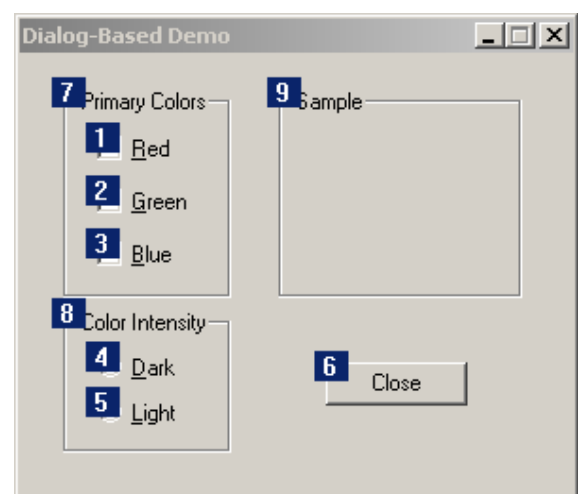
Настройка программы DlgDemo

Добавим в диалоговое окно программы элементы управления и напишем код их поддержки.

- Вызовем редактор диалоговых окон для ресурса с идентификатором IDD_DLGDEMO_DIALOG.
- Вызовем диалоговое окно Dialog Properties, откроем вкладку Styles. Установим флажки Title bar, System menu, Minimize box. Установки Style и Border оставим без изменения (Pop-up и Dialog Frame).
- В диалоговом окне удалим кнопку OK и надпись "TODO". Изменим свойства кнопки Cancel так, чтобы она называлась Close (на вкладке General).
- Добавим в диалоговое окно элементы управления, как показано на рисунке. Их свойства перечислены в таблице.

Идентификатор	Тип элемента управления	Надпись	Свойства, не задаваемые по умолчанию
IDC_STATIC	Рамка	Primary Colors	—
IDC_RED	Флажок	&Red	Group, Tab Stop
IDC_GREEN	Флажок	&Green	—
IDC_BLUE	Флажок	&Blue	—
IDC_STATIC	Рамка	Color Intensity	—
IDC_DARK	Переключатель	&Dark	Group, Tab Stop
IDC_LIGHT	Переключатель	&Light	—
IDC_SAMPLE	Рамка	Sample	—

- В меню Layout выберем команду Tab Order и зададим порядок обхода элементов управления, как показано на рисунке.
- С помощью мастера ClassWizard определим переменные для некоторых элементов управления. Запустим мастера, откроем вкладку Member Variables, выберем класс CDlgDemoDlg в списке Class name и добавим перечисленные ниже переменные, принимая стандартные категорию и тип.



Идентификатор	Имя переменной	Категория	Тип
IDC_RED	m_Red	Value	BOOL
IDC_GREEN	m_Green	Value	BOOL
IDC_BLUE	m_Blue	Value	BOOL
IDC_DARK	m_Intensity	Value	int

- С помощью мастера ClassWizard добавим обработчики сообщений для некоторых элементов управления. Во вкладке Message Maps добавим обработчик сообщения BN_CLICKED для следующих элементов управления: IDC_RED, IDC_GREEN, IDC_BLUE, IDC_DARK и IDC_LIGHT. Для каждой функции примем предлагаемое по умолчанию имя.

В файле DlgDemoDlg.cpp добавим код в сгенерированные обработчики (для остальных функций код аналогичен приведенному):

```
void CDlgDemoDlg::OnRed()
{
    m_Red = IsDlgButtonChecked (IDC_RED);
    InvalidateRect (&m_RectSample);
    UpdateWindow ();
}
void CDlgDemoDlg::OnDark()
{
    if (IsDlgButtonChecked (IDC_DARK))
    {
        m_Intensity = INT_DARK;
        InvalidateRect (&m_RectSample);
        UpdateWindow ();
    }
}
```

- В файле DlgDemoDlg.cpp добавим код в сгенерированный обработчик сообщения OnInitDialog добавим строки:

```
BOOL CDlgDemoDlg::OnInitDialog()
{
    // ...
    // TODO: добавьте код инициализации

    GetDlgItem (IDC_SAMPLE)->GetWindowRect (&m_RectSample);
    ScreenToClient (&m_RectSample);
    int Border = (m_RectSample.right - m_RectSample.left) / 8;
    m_RectSample.InflateRect (-Border, -Border);
}
```

```

        return TRUE; // возвращает TRUE, если фокус не установлен
                      // на элементе управления
    }

```

Здесь определяются координаты прямоугольника, в котором выводится цветной образец, и сохраняются в переменной `m_RectSample`. Функция `InflateRect` уменьшает размер прямоугольника, чтобы между ним и рамкой `Sample` оставалось свободное пространство.

- В файле `DlgDemoDlg.cpp` изменим присваивание начального значения переменной `m_Intensity` в конструкторе класса диалогового окна:

```

CDlgDemoDlg::CDlgDemoDlg(CWnd* pParent /*=NULL*/)
: CDialog(CDlgDemoDlg::IDD, pParent)
{
    //{{AFX_DATA_INIT(CDlgDemoDlg)
    m_Red = FALSE;
    m_Green = FALSE;
    m_Blue = FALSE;
    m_Intensity = INT_LIGHT;
    //}}AFX_DATA_INIT
    // Обратите внимание, что LoadIcon не требует в Win32
    // последующего вызова DestroyIcon
    m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
}

```

Отметим здесь, что редактирование кода, сгенерированного мастером `ClassWizard`, – это скорее исключение из правил.

- В файле `DlgDemoDlg.cpp` удалим вызов функции `CDialog::OnPaint` внутри оператора `else` функции `OnPaint` и заменим его следующим кодом:

```

void CDlgDemoDlg::OnPaint()
{
    // ...
    else
    {
        // вызов CDialog::OnPaint() был удален
        COLORREF Color = RGB
            (m_Red ? (m_Intensity==INT_DARK ? 128 : 255) : 0,
             m_Green ? (m_Intensity==INT_DARK ? 128 : 255) : 0,
             m_Blue ? (m_Intensity==INT_DARK ? 128 : 255) : 0);
        CBrush Brush (Color);
        CPaintDC dc(this);
        dc.FillRect (&m_RectSample, &Brush);
    }
}

```

Здесь код, сгенерированный AppWizard, в операторе if выводит значок программы, когда ее окно минимизировано, а код в else закрашивает прямоугольник в рамке

- В файле DlgDemoDlg.h в начале объявления класса CDlgDemoDlg добавим определение переменной m_RectSample и перечисления для задания интенсивности цвета:

```
class CDlgDemoDlg : public CDialog
{
public:
    CRect m_RectSample;
    enum {INT_DARK, INT_LIGHT};
    // ...
}
```

Функция InitInstance

InitInstance, сгенерированная мастером AppWizard *вместо* создания шаблона документа и вызова ProcessShellCommand для обработки командных строк, делает в нашем случае следующее:

- Создает объект класса CDlgDemoDlg;
- Вызывает для него функцию CDialog::DoModal. Проанализировав возвращаемое ею значение, можно при необходимости обработать значения переменных класса диалогового окна;
- Возвращает значение FALSE, чтобы программа завершила работу. В предыдущих наших программах она возвращала TRUE.

12.2. Программы просмотра форм

Программа просмотра формы имеет ту же базовую архитектуру, что и большинство рассмотренных нами ранее программ, однако ее класс представления порождается от класса CFormView, а не от классов представления MFC. Поэтому ее окно представления отображает совокупность элементов управления, а не пустую рабочую область. Элементы размещаются по шаблону диалогового окна.

В отличие от простой диалоговой программы программа просмотра формы:

- Имеет четыре стандартных MFC-класса: приложения, главного окна, документа и представления, а значит, годится для работы с документом.
- Имеет главное окно со стандартными компонентами окна, у которого можно изменять размеры, минимизировать и максимизировать его.
- Может иметь меню, панель инструментов, строку состояния.
- Так как класс CFormView порожден от CScrollView, при необходимости окно представления имеет горизонтальную и вертикальную полосы прокрутки.

Далее в качестве примера разрабатывается программа просмотра формы FormDemo, функционально повторяющая программу DlgDemo.

Генерация исходных файлов

- Запустим создание нового проекта. В качестве типа проекта выберем "MFC AppWizard (exe)". В качестве имени введем FormDemo.
- В диалоговом окне Step 1 установим опции Single document и Document/View architecture support (но не Dialog based).
- Выполним те же шаги со 2 по 5, что и для программы WinGreet.

В диалоговом окне Step 6 выберем класс представления CFormDemoView, а в качестве базового класса для него установим CFormView.

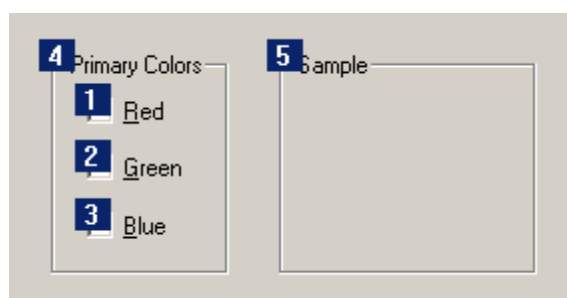
После генерации исходных файлов AppWizard автоматически откроет редактор диалоговых окон.

Настройка программы FormDemo

- Первоначально созданный шаблон диалогового окна содержит только надпись "TODO". Удалим ее. Свойства окна оставим без изменения.
- Добавим элементы управления со свойствами, приведенными в таблице. Можно для скорости просто скопировать их из программы DlgDemo.

Идентификатор	Тип элемента управления	Надпись	Свойства, не задаваемые по умолчанию
IDC_STATIC	Рамка	Primary Colors	—
IDC_RED	Флажок	&Red	Group, Tab Stop
IDC_GREEN	Флажок	&Green	—
IDC_BLUE	Флажок	&Blue	—
IDC_SAMPLE	Рамка	Sample	—

- Установим порядок обхода элементов управления, как показано на рисунке



- Для флажков IDC_RED, IDC_GREEN и IDC_BLUE заведем переменные с именами m_Red, m_Green, m_Blue с типом, установленным по умолчанию.
- Отредактируем меню (IDR_MAINFRAME). Удалим меню Edit и переименуем заголовок File на &Options. Удалим все команды в меню Options, кроме Exit и разделителя над ней, а также добавим пункты &Light Colors (идентификатор ID_OPTIONS_LIGHT) и &Dark Colors (идентификатор ID_OPTIONS_DARK).
- В классе CFormDemoView сгенерируем обработчики сообщений COMMAND и UPDATE_COMMAND_UI для команд меню ID_OPTIONS_LIGHT и ID_OPTIONS_DARK, а также обработчик сообщения BN_CLICKED для флагов IDC_RED, IDC_GREEN и IDC_BLUE.

В файле FormDemoView.cpp добавим приведенный ниже код в сгенерированные обработчики сообщений. Они приведены не все – оставшиеся имеют аналогичный код. Обратите внимание, что для пометки пункта меню используется SetRadio (отметка переключателя), а не SetCheck (просто флажок).

```
void CFormDemoView::OnUpdateOptionsLight (CCmdUI* pCmdUI)
{
    // TODO: Добавьте собственный код обработчика
    pCmdUI->SetRadio (m_Intensity == INT_LIGHT);
}
```

```
void CFormDemoView::OnOptionsLight ()
{
    // TODO: Добавьте собственный код обработчика
    m_Intensity = INT_LIGHT;
    CClientDC ClientDC (this);
    OnPrepareDC (&ClientDC);
    CRect Rect = m_RectSample;
    ClientDC.LPtoDP (&Rect);
    InvalidateRect (&Rect);
    UpdateWindow ();
}
```

```
void CFormDemoView::OnRed ()
{
    // TODO: Добавьте собственный код обработчика
    m_Red = IsDlgButtonChecked (IDC_RED);
    CClientDC ClientDC (this);
    OnPrepareDC (&ClientDC);
    CRect Rect = m_RectSample;
```

```

ClientDC.LPtoDP (&Rect);
InvalidateRect (&Rect);
UpdateWindow ();
}

```

Как и в программе DlgDemo, обработчики объявляют недействительной часть окна для перерисовки, однако они сначала преобразовывают координаты области из логических в координаты устройства. Это необходимо для того, чтобы окно правильно выводилось в случае наличия прокрутки.

Сгенерируем для класса представления CFormDemoView переопределенную версию функции OnDraw и модифицируем ее код следующим образом:

```

void CFormDemoView::OnDraw(CDC* pDC)
{
    // TODO: Добавьте код отображения собственных данных

    COLORREF Color = RGB
        (m_Red ? (m_Intensity==INT_DARK ? 128 : 255) : 0,
         m_Green ? (m_Intensity==INT_DARK ? 128 : 255) : 0,
         m_Blue ? (m_Intensity==INT_DARK ? 128 : 255) : 0);
    CBrush Brush (Color);
    pDC->FillRect (&m_RectSample, &Brush);
}

```

- Сгенерируем для класса представления CFormDemoView переопределенную версию функции OnInitialUpdate и модифицируем ее код следующим образом:

```

void CFormDemoView::OnInitialUpdate()
{
    CFormView::OnInitialUpdate();
    GetParentFrame()->RecalcLayout();
    ResizeParentToFit();

    GetDlgItem (IDC_SAMPLE)->GetWindowRect (&m_RectSample);
    ScreenToClient (&m_RectSample);
    int Border = (m_RectSample.right - m_RectSample.left) / 8;
    m_RectSample.InflateRect (-Border, -Border);
}

```

Эта функция вызывается непосредственно перед *первым* отображением документа в окне представления. Вызов функции RecalcLayout объекта главного окна нужен для размещения всех управляющих панелей программы, а чтобы главное окно отображало содержимое окна представления, вызывается функция CScrollView::ResizeParentToFit. Добавленный далее код помещен именно в эту функцию (а не в OnInitDialog), потому что

окно представления не получает сообщения WM_INITDIALOG. Отметим также, что поскольку к данному моменту окно еще не было прокручено, то в переменной m_RectSample можно сохранить координаты устройства, которые пока совпадают с логическими.

- Добавим переменные и определения констант в описание класса представления (файл FormDemoView.h):

```
class CFormDemoView : public CFormView
{
public:
    CBrush m_DialogBrush;
    int m_Intensity;
    CRect m_RectSample;

    enum {INT_DARK, INT_LIGHT};
    // ...
```

- В файле FormDemoView.cpp добавим инициализацию переменной m_Intensity:

```
CFormDemoView::CFormDemoView() : CFormView(CFormDemoView::IDD)
{
    // ...
    m_Intensity = INT_LIGHT;
}
```

- Зададим строку заголовка главного окна в функции InitInstance (файл FormDemo.cpp):

```
// ...
m_pMainWnd->SetWindowText ("Form-View Demo");
return TRUE;
}
```

Протестируем работу приложения.

Тема 13. Создание многодокументных приложений

13.1. Многодокументный интерфейс

Разработанные ранее программы являлись SDI-приложениями (SDI – Single Document Interface). В отличие от них программы с *многодокументным интерфейсом* (MDI-приложения, MDI – Multiple Document Interface) предназначены для работы с несколькими документами одновременно. Каждый из них просматривается в отдельном дочернем окне, находящемся в рабочей области внутри главного окна программы.

Мы разработаем MDI-версию программы MiniEdit, которая будет обладать следующими чертами:

- При первом запуске программы в дочернем окне будет открыт новый пустой документ.
- Если в меню File выбирается команда New или Open..., то новый документ отобразится в *отдельном дочернем окне*, не заменяя открытый ранее. У этих окон можно изменять размеры, минимизировать, максимизировать и упорядочивать в виде каскадной или мозаичной структуры.
- Перейти в необходимое дочернее окно можно либо щелчком мыши на нем, либо через меню Windows, либо перебирая их нажатием Ctrl+Tab или Ctrl+F6 (*следующее* окно), либо Ctrl+Shift+Tab (*предыдущее*).
- Заккрыть окно можно командой Close меню File. Команды сохранения и печати меню File, и команды меню Edit воздействуют только на активное дочернее окно.

13.2. Создание MDI-программы в среде Developer Studio

Генерация кода

С помощью AppWizard сгенерируем исходный код программы MiniEdit так же, как и для программы WinGreet, за следующими исключениями:

- В диалоговом окне Step 1 выберем опцию Multiple document;
- В диалоговом окне Step 4 щелкнем по кнопке Advanced... и в открывшемся диалоговом окне откроем вкладку Document Template Strings. Введем в поле File extension стандартное расширение файла – txt, в поле Doc type name (оно дублируется в File new name) – строку Text (для задания стандартных имен новым документам – Text1, Text2 и т.д.), наконец, в поле Filter name – строку Text Files (*.txt).
- В диалоговом окне Step 6 выберем имя класса представления CMiniEditView, а в списке Base Class – CEditView.

Замечание: В программах с многодокументным интерфейсом можно так же, как и в рассмотренных ранее примерах, добавить панель инструментов, строку состояния, диалоговую панель или средства прокрутки и разделения окон. В последнем случае каждое дочернее окно будет содержать собственную полосу прокрутки и вешку разбивки.

Файлы и классы MDI-приложения, создаваемые AppWizard, в целом аналогичны таковым для SDI-приложения, дополнительно используется класс дочернего масштабируемого окна.

13.3. Основные классы MDI-программы

Класс приложения

Управляет программой в целом. Для инициализации программы использует функцию `InitInstance`. В нашем примере класс имеет имя `CMiniEditApp`, файл заголовков – `MiniEdit.h`, файл реализации – `MiniEdit.cpp`.

Класс документа

Хранит данные документа и выполняет ввод/вывод. Программа создает *отдельные экземпляры* этого класса для каждого открытого документа. В нашем примере класс имеет имя `CMiniEditDoc`, файл заголовков – `MiniEditDoc.h`, файл реализации – `MiniEditDoc.cpp`.

Класс главного окна

Управляет главным окном программы. Порождается от класса `CMDIFrameWnd` (потомка `CFrameWnd`). В MDI-программах главное окно содержит *не* единственное окно представления, а общую рабочую область приложения, где находятся *дочерние масштабируемые окна* для каждого открытого документа. Каждое из них имеет свое окно представления. Поскольку главное окно содержит общую рабочую область, а не один открытый документ, его класс не включается в шаблон документа программы.

В отличие от SDI-приложений главное окно не создается автоматически при открытии первого документа, поэтому его создает и отображает функция `InitInstance`:

```
// Создание главного окна MDI-приложения
CMainFrame* pMainFrame = new CMainFrame;
if (!pMainFrame->LoadFrame(IDR_MAINFRAME))
    return FALSE;
m_pMainWnd = pMainFrame;
// ...
// Главное окно приложения отображается и обновляется
pMainFrame->ShowWindow(m_nCmdShow);
pMainFrame->UpdateWindow();
// ...
```

Сначала создается экземпляр класса `CMainFrame` для главного окна. Затем функция `LoadFrame` класса `CMainFrame` создает собственно главное окно, используя ресурс с идентификатором `IDR_MAINFRAME` (меню,

таблица горячих клавиш, строка заголовка и значок). Дескриптор окна сохраняется в переменной `m_pMainWnd` класса `CWinApp`. Вызов `CWnd::ShowWindow` делает окно видимым, а `CWnd::UpdateWindow` приводит к перерисовке рабочей области окна. В нашем примере класс главного окна имеет имя `CMainFrame`, файл заголовков – `MainFrm.h`, файл реализации – `MainFrm.cpp`.

Класс дочернего окна

В MDI-приложениях каждое дочернее окно содержит окно представления для отображения открытого документа. В программе `MiniEdit` класс дочернего окна имеет имя `CChildFrame`, файл заголовков – `ChildFrm.h`, файл реализации – `ChildFrm.cpp`.



Так как класс `CChildFrame` используется для создания и управления окном каждого документа, функция `InitInstance` включает этот класс в шаблон документа программы (вместо класса главного окна в SDI-приложении):

```
// Регистрация шаблонов документов приложений. Шаблоны служат
// для связи документа с главным окном и окном представления
CMultiDocTemplate* pDocTemplate;
pDocTemplate = new CMultiDocTemplate(
    IDR_TEXTTYPE,
    RUNTIME_CLASS(CMiniEditDoc),
    RUNTIME_CLASS(CChildFrame), // настройка дочернего окна
    RUNTIME_CLASS(CMiniEditView));
AddDocTemplate(pDocTemplate);
```

Замечание: Шаблон принадлежит классу `CMultiDocTemplate`, а не `CSingleDocTemplate`. Ему назначается идентификатор `IDR_TEXTTYPE` для ресурсов (меню, строка описания и значок). В частности меню

IDR_TEXTTYPE отображается каждый раз, когда открыт один или несколько документов. Строка IDR_TEXTTYPE содержит стандартное расширение файла документа и описание типа документа, значок IDR_TEXTTYPE отображается в каждом дочернем окне.

Использование документов различных типов

При помощи MFC можно писать программы, позволяющие работать с документами разных типов. Не рассматривая этот вопрос подробно, перечислим основные действия, позволяющие приложению управлять документами различных типов.

- Определите новый класс документа для управления его данными.
- Определите новый класс представления. Лучше взять за образец определения существующих классов, созданных посредством AppWizard.
- Создайте набор ресурсов для документа нового типа: меню, таблицу горячих клавиш (для SDI-приложения), строку и значок. Все они должны иметь один и тот же идентификатор. Строка должна иметь тот же формат, что и строка, созданная посредством AppWizard для документа первоначального типа.
- В функции `InitInstance` класса приложения создайте объект-шаблон (объект `CSingleDocTemplate` или `CMultiDocTemplate`), указав идентификатор новых ресурсов, новые классы документа и приложения, а также класс окна (класс главного окна для SDI-приложения или класс дочернего окна для MDI-приложения).
- Кроме того, в функции `InitInstance` вызовите функцию `AddDocTemplate`, чтобы добавить объект шаблона в объект приложения.

Если добавить в программу документы более, чем одного типа, то команда `New` в меню `File` будет отображать окно, позволяющее выбрать тип документа.

Класс представления

В программе `MiniEdit` класс представления `CMiniEditView` порождается от класса `CEditView`, файл его описания имеет имя `MiniEditView.h`, а файл реализации – `MiniEditView.cpp`.

Сгенерированный код программы

При порождении класса представления от класса `CEditView` мастер AppWizard генерирует код для чтения и записи текстовых документов из файлов и добавляет его в функцию `Serialize` класса документа (`CMiniEditDoc`). В отличие от SDI-версии в MDI-программе не требуется добавлять вызов `DeleteContents` в текст функции `Serialize`, так как для нового документа создается новое окно представления.

Кроме того, при создании кода AppWizard добавляет в функцию InitInstance класса приложения вызов DragAcceptFiles, позволяющий открыть файл путем перетаскивания его значка в окно программы.

Замечание.

Далее к программе добавляются средства, ранее включенные в SDI-версию. Это приходится делать заново, так как с помощью AppWizard невозможно переделать SDI-приложение в MDI (*вручную – можно*). Поэтому при разработке программы необходимо предварительное планирование для включения всех необходимых средств уже при генерации кода. Так, например, если программа создана без панели инструментов, то потом с помощью AppWizard ее уже не добавить.

13.4. Настройка ресурсов

Откроем раздел Menu в редакторе ResourceView. Так как мы сгенерировали MDI-приложение, там присутствуют два идентификатора: IDR_MAINFRAME и IDR_TEXTTYPE. Первый соответствует меню, отображаемому, когда все документы закрыты, второе – идентификатор меню, отображаемого, если открыт хотя бы один документ. Отредактируем меню IDR_TEXTTYPE, добавив в меню File непосредственно под командой Save As... разделитель и команду Print...:

Идентификатор	Надпись	Другие свойства
–	–	Separator
ID_FILE_PRINT	&Print...\tCtrl+P	–

Отредактируем меню Edit, добавив под командой Paste команду Select All, разделитель и команды Find..., Find Next и Replace...:

Идентификатор	Надпись	Другие свойства
ID_EDIT_SELECT_ALL	Select &All	–
–	–	Separator
ID_EDIT_FIND	&Find...	–
ID_EDIT_REPEAT	Find &Next\tF3	–
ID_EDIT_REPLACE	&Replace...	–

Команда New Window

При генерации кода в меню Window была помещена команда New Window, создающая дополнительное дочернее окно и окно представления, используемое для отображения документа в активном окне. Эта команда

предоставляет пользователю возможность просмотра и редактирования одного документа в нескольких окнах представления. Мы удалим эту команду, поскольку наше окно представления (порожденное от CEditView) само хранит документ, и было бы сложно эффективно обновлять несколько окон при внесении изменений в одно.

Если бы данные хранились в отдельном классе документа, то можно было бы создавать несколько его представлений (с помощью вешки разбивки или через команду New Window). В этом случае при изменении документа в одном из окон представления объект класса представления вызывает функцию UpdateAllViews класса документа для обновления остальных представлений.

Добавление горячих клавиш и значка

Добавим комбинации клавиш для команд меню Print... и Find Next:

Идентификатор	Клавиша
ID_FILE_PRINT	Ctrl+P
ID_EDIT_REPEAT	F3

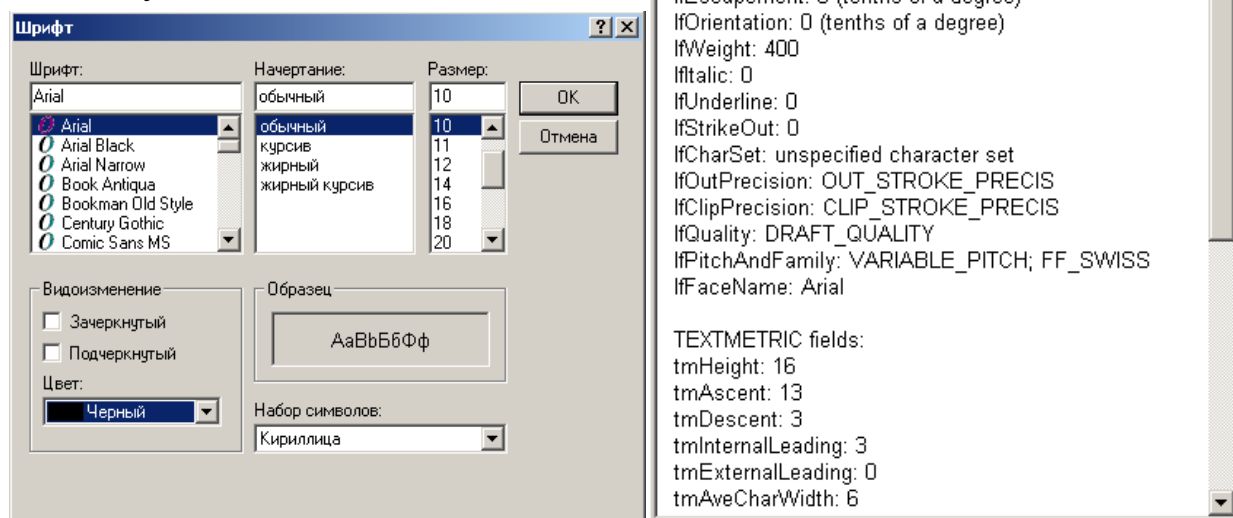
Если требуется изменить один или оба значка программы, следует иметь в виду, что идентификатор IDR_MAINFRAME соответствует главному окну программы, а идентификатор IDR_TEXTTYPE – каждому дочернему окну.

Тема 14. Ввод/вывод символов

14.1. Отображение текста

В данной теме мы рассмотрим вопрос об отображении строк текста внутри окна представления. Как пример будет создана программа TextDemo.

Она должна вызывать стандартное диалоговое окно Font для выбора шрифта, его стиля, цвета и эффектов. После выбора установок шрифта программа выводит информацию о шрифте, используя все перечисленные установки.



Генерация исходного кода программы

Для генерации исходных файлов воспользуемся мастером AppWizard. Выполним те же шаги, что и для генерации программы WinGreet, задав в качестве имени программы TextDemo. На шаге 6 для класса представления CTextDemoView выберем в качестве базового класса класс CScrollView.

Отображение текста в окне представления

Собственно текст сохраняется обычно в классе документа (код будет создан далее), а здесь мы напишем код, необходимый для вывода текста в окне представления.

Прежде всего, зададим расстояние от верхней левой границы текста до границ окна представления (одинаковое) в файле TextDemoView.h:

```
const int MARGIN = 10; // расстояние от текста до верхней и
                        // левой границы окна представления
class CTextDemoView : public CScrollView
{
    // ...
```

В файле TextDemoView.cpp добавим следующий код в функцию OnDraw:

```
void CTextDemoView::OnDraw(CDC* pDC)
{
    CTextDemoDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    // TODO: Добавьте код отображения собственных данных

    // возврат, если шрифт не создан:
    if (pDoc->m_Font.m_hObject == NULL)
        return;

    RECT ClipRect;
    int LineHeight;
    TEXTMETRIC TM;
    int Y = MARGIN;

    // выбор шрифта в объект контекста устройства:
    pDC->SelectObject (&pDoc->m_Font);

    // получение метрики текста:
    pDC->GetTextMetrics (&TM);
    LineHeight = TM.tmHeight + TM.tmExternalLeading;

    // установка атрибутов текста:
    pDC->SetTextColor (pDoc->m_Color);
    pDC->SetBkMode (TRANSPARENT);

    // получение координат недействительной области:
    pDC->GetClipBox (&ClipRect);

    // отображение строки заголовка:
    pDC->TextOut (MARGIN, Y, "FONT PROPERTIES");
    // отображение строк текста:
    for (int Line = 0; Line < NUMLINES; ++Line)
    {
        Y += LineHeight;
        if (Y + LineHeight >= ClipRect.top &&
            Y <= ClipRect.bottom)
            pDC->TextOut (MARGIN, Y,
                          pDoc->m_LineTable [Line]);
    }
}
```

Основные этапы отображения текста внутри окна представления

- Если отображение производится *не функцией OnDraw*, сначала необходимо получить (создать) объект контекста для окна представления. Пример таких действий будет приведен в следующей программе.
- Выбрать шрифт, если не хотите использовать стандартный шрифт System. В нашем примере Функции CDC::SelectObject передается адрес объекта шрифта, содержащего его полное описание. Эта функция может использоваться и для выбора других объектов, влияющих на отображение графики.
- Если требуется, установите размеры текста.
- Если требуется, установите желаемые атрибуты текста.
- Если текст отображается функцией OnDraw, получите размеры недействительной области окна представления (т.е. подлежащей перерисовке).
- Вызовите соответствующую функцию класса CDC для отображения текста (если эта функция – OnDraw, отобразите только часть текста, попадающую в недействительную область).

Метрики шрифта

В приведенном выше примере для вычисления расстояния между строками использовались значения, записанные в поля переменной ТМ (тип – структура TEXTMETRIC), а заполнялась она через вызов функции CDC::GetTextMetrics. Ниже объясняется смысл характеристик символа, получаемых через этот вызов.



Цвет шрифта

В нашем примере устанавливается цвет текста, сохраненный в переменной `m_Color` класса документа. Передача функции `SetBkMode` значения `TRANSPARENT` означает, что символы отображаются поверх существующих цветов на отображающей поверхности устройства. Если бы было использовано значение `OPAQUE`, то в качестве цвета фона использовался бы цвет, установленный функцией `CDC::SetBkColor` (по умолчанию – белый).

Функции класса CDC для установки и определения атрибутов текста

Функция	Назначение
SetBkColor	Определяет цвет фона текста
SetBkMode	Разрешает или запрещает окраску фона текста
SetMapMode	Устанавливает текущий режим отображения (система координат и единицы измерения для позиционирования)
SetTextAlign	Определяет выравнивание текста
SetTextCharacterExtra	Задаёт величину межсимвольного интервала (разреженного или уплотнённого)
SetTextColor	Определяет цвет шрифта
GetBkColor	Возвращает цвет фона текста
GetBkMode	Возвращает режим для фона текста
GetMapMode	Возвращает текущий режим отображения
GetTextAlign	Возвращает стиль выравнивания текста
GetTextCharacterExtra	Возвращает количество лишних межсимвольных интервалов
GetTextColor	Возвращает цвет шрифта

Функции отображения текста

В приведенном примере мы для вывода текста воспользовались функцией `TextOut`, которая в качестве аргументов получает координаты верхнего левого угла первого символа в строке, а также собственно выводимую строку. Ниже перечислены другие функции класса CDC для вывода текста.

Функция	Назначение
DrawText	Отображение текста в заданном прямоугольнике. Используется для изменения отступа табуляции, выравнивания и центрирования текста, а также для разрыва строк между словами для их подгонки к размерам прямоугольника
ExtTextOut	Отображение текста в заданном прямоугольнике. Используется для усечения текста, который не попадает в прямоугольник, заполнения прямоугольника цветом фона или изменения интервала между символами.
GrayString	Вывод затененного текста. Обычно используется для указания недоступных опций или пунктов.
TabbedTextOut	Работает подобно функции <code>TextOut</code> , но увеличивает отступ табуляции с использованием заданного шага.
TextOut	Отображает строку с заданной начальной позиции

Создание объекта Font и сохранение текста

Здесь мы добавим средства для отображения стандартного диалогового окна Font, кодом инициализации объекта шрифта, а также строками для генерации и сохранения текста.

Откроем меню IDR_MAINFRAME, удалим меню File и Edit, а затем добавим слева от Help новое меню Options со следующими пунктами:

Идентификатор	Надпись	Другие свойства
–	&Options	Pop-up
ID_OPTIONS_FONT	&Font...	–
–	–	Separator
ID_APP_EXIT	E&xit	–

При необходимости можно также отредактировать значок программы.

В диалоговом окне мастера ClassWizard создадим обработчика для добавленной команды Font. Для этого на вкладке Message Maps выберем класс CTextDemoDoc, пункт ID_OPTIONS_FONT из списка Object IDs и добавим обработчик сообщения COMMAND, согласившись с предложенным именем OnOptionsFont.

Сначала в файле TextDemoDoc.h добавим определения необходимых переменных и констант:

```
// ...
const int NUMLINES = 42; // количество строк, сохраненных
                          // в документе и отображенных
                          // в окне представления

class CTextDemoDoc : public CDocument
{
public:
    COLORREF m_Color;
    CString m_LineTable [NUMLINES];
    CFont m_Font;
// ...
```

Переменная m_Color предназначена для хранения цвета, выбранного в диалоговом окне Font, массив m_LineTable содержит собственно строки, а переменная m_Font MFC-класса CFont содержит объект шрифта, используемый для установки шрифта текста.

Далее в файле TextDemoDoc.cpp добавим собственно код обработчика (функция OnOptionsFont). Ввиду довольно большого объема этот код здесь не приводится (его можно взять из прилагаемого готового проекта и вставить в свой). Обработчик выполняет следующие действия:

- Отображает диалоговое окно Font.
- Передает описание шрифта в функцию CFont::CreateFontIndirect для инициализации объекта шрифта m_Font.
- Записывает описания шрифта в строки массива m_LineTable.
- Создает объект контекста устройства и выбирает в нем объект шрифта.
- Вызывает функцию GetTextMetrics объекта контекста устройства для получения свойств активного шрифта устройства (информация копируется в структуру TEXTMETRIC).
- Записывает информацию о свойствах активного шрифта в строки массива m_LineTable.
- Вызывает функцию UpdateAllViews для принудительного отображения функцией OnDraw класса представления строк текста из массива m_LineTable.

Использование стандартных шрифтов

Диалоговое окно Font позволяет выбрать *любой* из доступных шрифтов для экрана или другого устройства. Однако можно выбрать *стандартный шрифт* из небольшого набора типовых шрифтов Windows. Для этого достаточно выбрать функцию SelectStockObject класса CDC

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

Здесь nIndex – индекс требуемого шрифта. Пример использования:

```
void CTextDemoView::OnDraw(CDC* pDC)
{
    CTextDemoDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);

    // TODO: здесь добавьте код отображения
    pDC->SelectStockObject(SYSTEM_FIXED_FONT);
    // Установка атрибутов текста ...
    // Отображение текста в окне представления ...
}
```

Функция SelectStockObject также используется для выбора других стандартных инструментов, например, кистей и перьев, используемых при рисовании.

Значения *nIndex* для выбора стандартных шрифтов

Значение параметра	Стандартный шрифт
SYSTEM_FONT	Системный шрифт System с переменным питчем. Используется для отображения текста на экране, если шрифт не выбран в объекте контекста устройства
SYSTEM_FIXED_FONT	Системный шрифт с фиксированным питчем Fixedsys. Удобен для программ редактирования и других приложений, использующих шрифт с фиксированным питчем. Используется, например в Notepad
ANSI_VAR_FONT	Шрифт с переменным питчем, более мелкий, чем шрифт, заданный значением SYSTEM_FONT.
ANSI_FIXED_FONT	Шрифт с фиксированным питчем, более мелкий, чем шрифт, заданный значением SYSTEM_FIXED_FONT.
DEVICE_DEFAULT_FONT	Шрифт устройства, заданный по умолчанию. Например, для окна – System, для принтера HP Laser Jet II – Courier.
OEM_FIXED_FONT	Шрифт с фиксированным питчем Terminal. Соответствует набору символов, используемых основными аппаратными средствами. В ранних версиях Windows использовался для отображения текста в окне MS-DOS.

Поддержка средств прокрутки

На этапе генерации кода мы указали, что класс представления порождается от класса CScrollView, что позволяет ему поддерживать средства прокрутки.

Теперь нам нужно добавить код, сообщающий MFC текущий размер документа. AppWizard сгенерировал виртуальную функцию OnInitialUpdate со стандартным кодом, передающим размер документа. Ее необходимо *удалить*, так как размер документа теперь зависит от выбранных установок шрифта. Он у нас будет определяться в виртуальной функции OnUpdate, получающей управление при каждом вызове функции UpdateAllViews.

Запустим мастера ClassWizard, откроем вкладку Message Maps и для класса CTextDemoView удалим функцию OnInitialUpdate. Теперь для того

же класса в списке Message выберем OnUpdate и добавим функцию-обработчик.

Отредактируем ее код, как показано ниже.

```
void CTextDemoView::OnUpdate(CView* pSender, LPARAM lHint,
                             CObject* pHint)
{
    // TODO: Добавьте собственный код обработчика
    CTextDemoDoc* PDoc = GetDocument();

    if (PDoc->m_Font.m_hObject == NULL) // шрифт не создан
        SetScrollSizes (MM_TEXT, CSize (0,0));
    else // шрифт создан
    {
        CClientDC ClientDC (this);
        int LineWidth = 0;
        SIZE Size;
        TEXTMETRIC TM;

        ClientDC.SelectObject (&PDoc->m_Font);
        ClientDC.GetTextMetrics (&TM);
        for (int Line = 0; Line < NUMLINES; ++Line)
        {
            Size = ClientDC.GetTextExtent
                (PDoc->m_LineTable [Line],
                PDoc->m_LineTable [Line].GetLength ());
            if (Size.cx > LineWidth)
                LineWidth = Size.cx;
        }

        Size.cx = LineWidth + MARGIN;
        Size.cy = (TM.tmHeight + TM.tmExternalLeading) *
            (NUMLINES + 1) + MARGIN;
        SetScrollSizes (MM_TEXT, Size);
        ScrollToPosition (CPoint (0, 0));
    }
    CScrollView::OnUpdate (pSender, lHint, pHint);
}
```

Функция OnUpdate вычисляет полный размер основной части документа, исходя из размеров установленного шрифта, и устанавливает его, вызывая функцию SetScrollSize.

Если шрифт не выбран, то задается нулевой размер документа, а полоса прокрутки автоматически скрывается. Если шрифт выбран, вычисляется общая высота и ширина текста, а затем размеры передаются функции SetScrollSize.

Сначала создается объект контекста устройства окна представления, затем выбирается объект шрифта в объекте контекста устройства и вызывается функция `GetTextMetrics` для получения размеров символов. Здесь эти размеры используются только для вычисления высоты, так как функция `GetTextMetrics` возвращает *среднюю* ширину символа (в поле `tmAveCharWidth` структуры `TEXTMETRIC`). Для вычисления длины строки используется функция `CDC::GetTextExtent`, которая позволяет правильно определить ширину строки даже для шрифтов с переменным питчем.

Далее идет вызов функции `CScrollView::ScrollToPosition` для того, чтобы прокрутить окно представления к началу текста.

Наконец, вызов функции `OnUpdate` класса `CScrollView` приводит к объявлению недействительным всего окна, что приводит к его перерисовке.

Для вывода заголовка программы в конце функции `CTextDemoApp::InitInstance` в файле `TextDemo.cpp` добавим вызов

```
m_pMainWnd->SetWindowText ("Text Demo");
```

14.2. Чтение кодов символов, вводимых с клавиатуры

Ниже мы добавим в код только что созданной программы средства, позволяющие осуществлять прокрутку окна представления с помощью клавиатуры.

Обработка сообщения `WM_KEYDOWN`

При каждом нажатии клавиши на клавиатуре система посылает сообщение `WM_KEYDOWN` окну, в котором в данный момент находится *фокус ввода*. В программе MFC, сгенерированной мастером `AppWizard`, при активном главном окне фокус содержится в окне представления (в MDI-приложении фокус находится в активном окне представления).

Замечание. Если нажата системная клавиша (`PrintScreen`, `Alt` или `Alt+клавиша`), то посылается сообщение `WM_SYSKEYDOWN`, а не `WM_KEYDOWN`. Эти сообщения обычно обрабатывает `Windows`.

Для добавления обработчика сообщения `WM_KEYDOWN` в программу `TextDemo` запустим `ClassWizard`, откроем вкладку `Message Maps`, в списке `Class name` выберем класс `CTextDemoView`, в списке `Object IDs` – `CTextDemoView`, а в списке `Message` – идентификатор `WM_KEYDOWN` и добавим функцию-обработчик (имя по умолчанию – `OnKeyDown`).

Отредактируем код вновь созданной функции (в файле `TextDemoView.cpp`) добавив в него следующие строки.

```

void CTextDemoView::OnKeyDown(UINT nChar, UINT nRepCnt,
                               UINT nFlags)
{
    // TODO: Добавьте собственный код обработчика
    CSize DocSize = GetTotalSize ();
    RECT ClientRect;
    GetClientRect (&ClientRect);

    switch (nChar)
    {
        case VK_LEFT:    // стрелка влево
            if (ClientRect.right < DocSize.cx)
                SendMessage (WM_HSCROLL, SB_LINELEFT);
            break;
        case VK_RIGHT:   // стрелка вправо
            if (ClientRect.right < DocSize.cx)
                SendMessage (WM_HSCROLL, SB_LINERIGHT);
            break;
        case VK_UP:      // стрелка вверх
            if (ClientRect.bottom < DocSize.cy)
                SendMessage (WM_VSCROLL, SB_LINEUP);
            break;
        case VK_DOWN:    // стрелка вниз
            if (ClientRect.bottom < DocSize.cy)
                SendMessage (WM_VSCROLL, SB_LINEDOWN);
            break;
        case VK_HOME:    // клавиша Home
            if (::GetKeyState (VK_CONTROL) & 0x8000)
                // Ctrl+Home
            {
                if (ClientRect.bottom < DocSize.cy)
                    SendMessage (WM_VSCROLL, SB_TOP);
            }
            else          // Home без Ctrl
            {
                if (ClientRect.right < DocSize.cx)
                    SendMessage (WM_HSCROLL, SB_LEFT);
            }
            break;
        case VK_END:     // клавиша End
            if (::GetKeyState (VK_CONTROL) & 0x8000)
                // Ctrl+End
            {
                if (ClientRect.bottom < DocSize.cy)
                    SendMessage (WM_VSCROLL, SB_BOTTOM);
            }
    }
}

```

```

else          // End без Ctrl
{
    if (ClientRect.right < DocSize.cx)
        SendMessage (WM_HSCROLL, SB_RIGHT);
}
break;
case VK_PRIOR: // клавиша PgUp
    if (ClientRect.bottom < DocSize.cy)
        SendMessage (WM_VSCROLL, SB_PAGEUP);
    break;
case VK_NEXT:  // клавиша PgDn
    if (ClientRect.bottom < DocSize.cy)
        SendMessage (WM_VSCROLL, SB_PAGEDOWN);
    break;
}
CScrollView::OnKeyDown(nChar, nRepCnt, nFlags);
}

```

О работе функции CTextDemoView::OnKeyDown

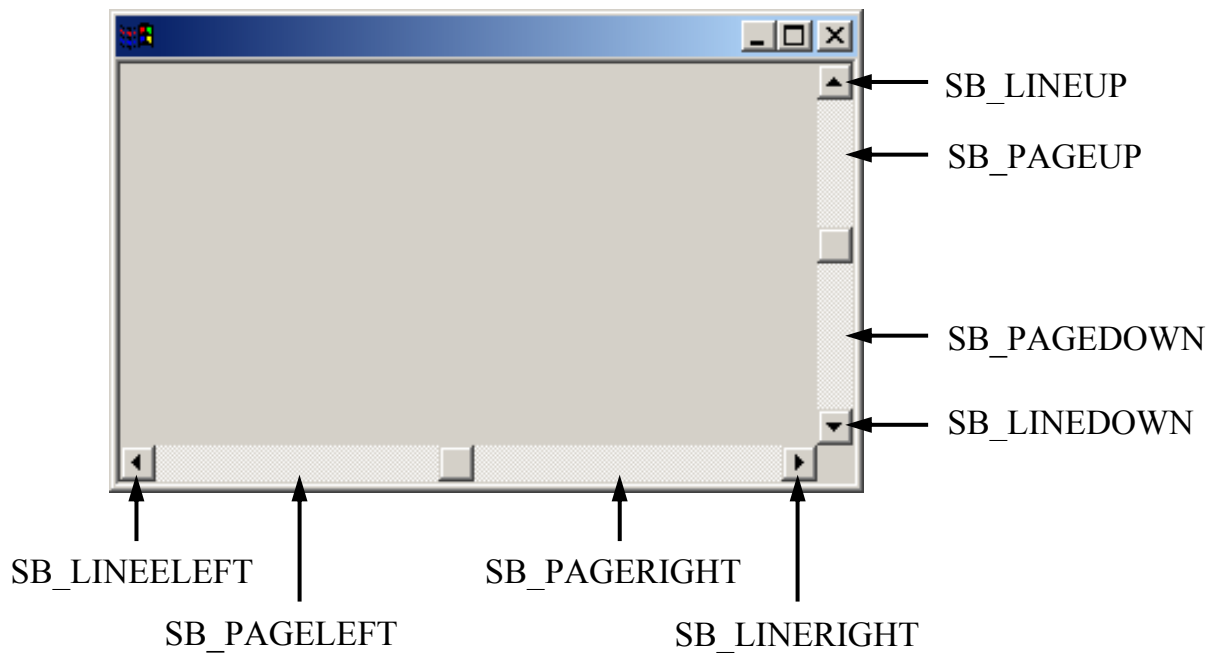
При нажатии большинства клавиш генерируется сообщение WM_CHAR, и удобнее обрабатывать именно его (поскольку обработчик получает код символа), однако это не относится к некоторым клавишам, в частности тем, нажатие которых обрабатывается нашей функцией OnKeyDown.

Первый аргумент функции OnKeyDown (nChar) содержит *виртуальный код клавиши*, который используется функцией для перехода к соответствующей подпрограмме. Нажатие следующих клавиш не генерирует сообщения WM_CHAR:

- Стрелки, а также цифра 5 на дополнительной клавиатуре (без Num Lock);
- Home, End, PgUp, PgDn, Insert, Delete;
- Shift, Ctrl, Pause, Caps Lock, Num Lock, Scroll Lock;
- Функциональные клавиши F1 – F12.

Для того, чтобы проверить, была ли нажата дополнительно клавиша Ctrl или Shift *во время генерации* сообщения WM_KEYDOWN, можно воспользоваться функцией ::GetKeyState. Текущее состояние клавиши *во время обработки* можно узнать, вызывая функцию ::GetAsyncKeyState.

Обработка нажатий клавиш состоит в генерации сообщений, идентичных сообщениям, передающимся при щелчках мышью в полосах прокрутки окна – они представлены на следующем рисунке. Обработчики для этих сообщений предоставляются классом CScrollView.



В приведенном выше примере кода использовались установки по умолчанию, при которых страница равна 1/10 размера документа, строка – 1/10 размера страницы. Эти установки можно изменить, вызвав функцию `CScrollView::SetScrollSizes`.

Отметим также, что до передачи сообщения полосе прокрутки функция `OnKeyDown` проверяет, видима ли соответствующая полоса прокрутки, иначе код `CScrollView` будет работать неправильно. MFC скрывает полосу, если окно представления имеет такой же (или больший размер – высоту или ширину), что и отображаемый текст. Для получения размеров текста вызывается функция `CScrollView::GetTotalSize`:

```
CSize DocSize = GetTotalSize ();
RECT ClientRect;
GetClientRect (&ClientRect);
```

За проверку соответственно ширины и высоты отвечают строки:

```
if (ClientRect.right < DocSize.cx)
// ...
if (ClientRect.bottom < DocSize.cy)
// ...
```

Обработка сообщения `WM_CHAR`

При нажатии большинства клавиш окну с фокусом ввода передается сообщение `WM_CHAR` (исключения были перечислены выше). Обработка этого сообщения более удобна для ввода текстовой информации, чем `WM_KEYDOWN`, поскольку обработчик получает через аргументы не виртуальный код клавиши, а стандартный код ANSI.

Здесь для иллюстрации базовых методов обработки сообщения `WM_CHAR` мы создадим программу `Echo`, которая просто выводит введенные символы на экран в виде строки (без переноса по достижении гра-

ницы окна). Для очистки строки предусмотрим команду Clear в пункте меню Edit.

Сгенерируем исходный код, используя AppWizard, выбирая те же установки, что и для программы WinGreet (за исключением имени проекта – Echo).

Отредактируем меню IDR_MAINFRAME, удалив меню File и все пункты в меню Edit. Добавим в меню Edit следующие пункты:

Идентификатор	Надпись	Другие свойства
–	&Edit	Pop-up
ID_EDIT_CLEAR	&Clear	–
–	–	Separator
ID_APP_EXIT	E&xit	–

В диалоговом окне мастера ClassWizard создадим обработчика для сообщений WM_CHAR. Для этого на вкладке Message Maps в списке Class name выберем CEchoView, в списке Object IDs также CEchoView и добавим обработчик сообщения WM_CHAR с именем OnChar.

Создадим также обработчик команды Clear, выбрав в списке Object IDs идентификатор ID_EDIT_CLEAR, а в списке Messages пункт COMMAND. Согласимся с предложенным именем функции-обработчика OnEditClear.

В файле EchoDoc.h добавим определение переменной m_TextLine в описании класса CEchoDoc:

```
class CEchoDoc : public CDocument
{
public:
    CString m_TextLine;
// Оставшаяся часть определения класса
```

Переменная m_TextLine предназначена для хранения вводимых символов.

В файле EchoView.cpp добавим код в сгенерированную функцию OnChar:

```
void CEchoView::OnChar(UINT nChar, UINT nRepCnt, UINT nFlags)
{
    // TODO: Добавьте собственный код обработчика

    if (nChar < 32)
    {
        ::MessageBeep (MB_OK); // генерация стандартного звука
        return;
    }
}
```

```

CEchoDoc* PDoc = GetDocument();
PDoc->m_TextLine += nChar;

CClientDC ClientDC (this);

ClientDC.SetTextColor (::GetSysColor (COLOR_WINDOWTEXT));
ClientDC.SetBkMode (TRANSPARENT);
ClientDC.TextOut (0, 0, PDoc->m_TextLine);

CView::OnChar(nChar, nRepCnt, nFlags);
}

```

Функция OnChar получает управление при каждом нажатии символьной клавиши, при этом параметр nChar содержит код соответствующего ANSI-символа.

В нашем примере мы игнорируем символы с кодами менее 32, хотя при написании, например, текстового редактора разумно на некоторые из них назначать определенные действия (скажем, на Backspace – код 8, или Enter – 13).

Введенный символ добавляется в конец строки m_TextLine посредством перегруженного оператора += класса CString.

Далее функция создает контекст устройства для окна представления и использует его для отображения всей строки. При отображении используется стандартный системный шрифт, в качестве цвета вызовом

```
ClientDC.SetTextColor (::GetSysColor (COLOR_WINDOWTEXT));
```

назначается цвет, установленный в Windows для вывода текста.

Замечание. Если окно представления поддерживает средства прокрутки, то *перед* отображением текста или графики необходимо передать созданный объект контекста устройства (экземпляр класса CClientDC) в функцию CView::OnPrepareDC для его согласования с текущей позицией прокрутки документа.

В файле EchoView.cpp добавим код в функцию OnDraw:

```

void CEchoView::OnDraw(CDC* pDC)
{
    CEchoDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    // TODO: Добавьте код отображения данных
    pDC->SetTextColor (::GetSysColor (COLOR_WINDOWTEXT));
    pDC->SetBkMode (TRANSPARENT);
    pDC->TextOut (0, 0, pDoc->m_TextLine);
}

```

Мы могли бы в функции OnChar вместо непосредственного отображения текста просто вызвать функцию CDocument::UpdateAllViews, однако это могло бы вызвать некоторые нежелательные эффекты, например, мер-

цание вследствие очистки недействительной области окна непосредственно перед вызовом OnDraw. Наконец, добавим код в функцию OnEditClear:

```
void CEchoView::OnEditClear()
{
    // TODO: Добавьте собственный код обработчика
    CEchoDoc* PDoc = GetDocument();
    PDoc->m_TextLine.Empty();
    PDoc->UpdateAllViews(NULL);
}
```

14.3. Управление курсором при редактировании

В созданной версии программы Echo отсутствует привычный мигающий курсор, отмечающий место вставки данных в текст. Для его добавки в программу выполним следующие действия:

Добавление новых функций обработки сообщений

В окне мастера ClassWizard откроем вкладку Message Maps, в списках Class name и Object IDs выберем имя класса CEchoView. Добавим три функции обработки сообщений (имена возьмем по умолчанию), выбрав в списке Message сообщения WM_CREATE, WM_KILLFOCUS и WM_SETFOCUS.

Добавим в определение класса CEchoView (файл EchoView.h) объявление трех новых переменных: m_CaretPos для хранения текущей позиции курсора и пары m_XCaret, m_YCaret – для хранения его размера.

```
class CEchoView : public CView
{
private:
    POINT m_CaretPos;
    int m_XCaret, m_YCaret;
    // Оставшаяся часть определения класса ...
```

Добавление кода функций-обработчиков

Добавим в реализацию класса CEchoView (файл EchoView.cpp) следующий код:

```
CEchoView::CEchoView()
{
    // TODO: Добавьте собственный код конструктора
    m_CaretPos.x = m_CaretPos.y = 0;
}
```

```

int CEchoView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CView::OnCreate(lpCreateStruct) == -1)
        return -1;
    // TODO: Добавьте здесь специальный код
    CClientDC ClientDC (this);
    TEXTMETRIC TM;
    ClientDC.GetTextMetrics (&TM);
    m_XCaret = TM.tmAveCharWidth / 3;
    m_YCaret = TM.tmHeight + TM.tmExternalLeading;
    return 0;
}

```

Функция OnCreate вызывается после первоначального создания окна представления, но перед тем, как оно станет видимым. В нашем примере размеры курсора (каретки) рассчитываются, исходя из текущего размера символов. Каретка имеет ширину, равную трети средней ширины символа и высоту, равную высоте символа с учетом вертикальных просветов.

Добавим в функцию OnSetFocus (файл EchoView.cpp) следующий код:

```

void CEchoView::OnSetFocus(CWnd* pOldWnd)
{
    CView::OnSetFocus(pOldWnd);
    // TODO: Добавьте собственный код обработчика
    CreateSolidCaret (m_XCaret, m_YCaret);
    SetCaretPos (m_CaretPos);
    ShowCaret ();
}

```

Функция OnSetFocus вызывается каждый раз, когда окно представления получает фокус ввода. Она вызывает функцию CreateSolidCaret для создания курсора, передавая его размеры. Затем вызывается функция CWnd::SetCaretPos, чтобы поместить курсор в нужную позицию, а затем функция CWnd::ShowCaret, чтобы его отобразить, так как вновь созданный курсор невидим. Отметим также, что при первом появлении окна представления функция OnSetFocus вызывается после OnCreate, что дает возможность использовать вычисленные там размеры курсора.

Понятно, почему курсор надо создавать при создании окна представления, однако почему его требуется создавать *каждый раз*, когда окно получает фокус? Потому что при потере фокуса курсор следует уничтожить, что мы и сделаем в функции OnKillFocus (файл EchoView.cpp) :

```

void CEchoView::OnKillFocus(CWnd* pNewWnd)
{
    CView::OnKillFocus(pNewWnd);
    // TODO: Добавьте собственный код обработчика
    ::DestroyCaret ();
}

```

Курсор должен быть уничтожен, потому что он является общедоступным ресурсом Windows. Одновременно в рабочей области окна Windows может отображаться только один курсор, и он должен отображаться внутри окна, обладающего фокусом ввода.

Добавим код для работы с курсором в функции OnChar и OnEditClear (файл EchoView.cpp) :

```
void CEchoView::OnChar(UINT nChar, UINT nRepCnt, UINT nFlags)
{
    // ...
    ClientDC.SetBkMode (TRANSPARENT);
    HideCaret ();
    ClientDC.TextOut (0, 0, PDoc->m_TextLine);
    CSize Size = ClientDC.GetTextExtent(PDoc->m_TextLine,
                                          PDoc->m_TextLine.GetLength ());
    m_CaretPos.x = Size.cx;
    SetCaretPos (m_CaretPos);
    ShowCaret ();
    CView::OnChar(nChar, nRepCnt, nFlags);
}
void CEchoView::OnEditClear()
{
    // ...
    m_CaretPos.x = 0;
    SetCaretPos (m_CaretPos);
}
```

В добавленном коде вызовы функций CWnd::HideCaret и CWnd::ShowCaret используются для того, чтобы сделать курсор невидимым или видимым соответственно. Для установки курсора на новое место сначала вызывается функция GetTextExtent, чтобы вычислить новую длину строки, а затем SetCaretPos, чтобы поместить курсор в конец строки.

Замечания.

1. Не следует скрывать курсор в коде функции OnDraw, так как Windows автоматически скрывает курсор до вызова этой функции и восстанавливает после возврата из нее.

2. Вызовы функций HideCaret кумулятивные, т.е. при нескольких вызовах вы должны столько же раз вызвать функцию ShowCaret для появления курсора.

Наконец, добавим в код функции InitInstance (файл Echo.cpp) для отображения имени программы в заголовке:

```
BOOL CEchoApp::InitInstance()
{
    // ...
    m_pMainWnd->SetWindowText ("Echo");
    return TRUE;
}
```

Тема 15. Использование функций рисования

В этой и следующей теме мы рассмотрим два различных подхода к созданию и манипулированию графическими изображениями.

В рамках данной темы будет рассмотрено использование функций рисования, предназначенных для создания рисунков, состоящих из отдельных геометрических фигур, таких как прямые линии, дуги и прямоугольники. В сочетании со средствами манипуляции растровыми изображениями, которые будут рассмотрены в следующей теме, они составляют полный набор для создания и обработки графических объектов внутри окна представления или иного устройства (например, принтера).

Основные вопросы, которые будут рассмотрены в данной теме:

- Создание объекта контекста устройства
- Выбор средств рисования внутри объекта
- Установка атрибутов рисования для объекта
- Создание графических изображений
- Функции рисования – члены класса CDC
- Пример использования – программа MiniDraw

15.1. Создание объекта контекста устройства

Для отображения текста или графики необходим объект контекста устройства, соответствующий окну или устройству вывода данных. Он сохраняет выбранные средства и установленные атрибуты и предоставляет необходимые функции-члены для отображения.

Для отображения графического объекта с помощью функции OnDraw класса представления она получает указатель на объект контекста устройства, в котором следует отобразить объект:

```
void CMyView::OnDraw(CDC* pDC)
{
    // Отобразите графику, используя указатель pDC
}
```

Функция OnDraw вызывается при рисовании или перерисовке окна представления. Если программа отображает графику не в окне представления, а каком-то другом окне (например, в диалоговом), то для класса окна необходимо добавить обработчик сообщения WM_PAINT (с именем OnPaint) и создать в нем объект контекста устройства, порождаемый от MFC-класса CPaintDC (пример был приведен в теме "Создание модальных диалоговых окон"):

```
void CMyDialog::OnPaint()
{
    CPaintDC PaintDC (this);
    // Отобразите графику, используя PaintDC
}
```

Для отображения графики в окне представления или другом окне из функции, которая *не обрабатывает* сообщение WM_PAINT, необходимо создать объект контекста устройства, порождаемый от MFC-класса CClientDC. Если объект поддерживает прокрутку, то перед его использованием нужно вызвать функцию CScrollView::OnPrepareDC для настройки объекта на текущую позицию документа.

```
void CMyView::OtherFunction()
{
    CClientDC ClientDC (this);

    // Если графика отображается в окне представления,
    // поддерживающем прокрутку
    OnPrepareDC (&ClientDC);

    // Отобразите графику, используя ClientDC
}
```

Функции рисования, рассмотренные ниже, являются членами класса CDC (базового для остальных классов контекста устройства), следовательно, их можно использовать при работе с объектами контекста устройства произвольного типа, в том числе принтерами и плоттерами.

15.2. Выбор средств рисования внутри объекта

Перо и кисть

Перо и кисть – два инструмента, выбор которых отражается на работе функций класса CDC. Они относятся к объектам (в смысле структурам данных Windows) GDI – *graphic device interface*. Другие графические объекты: шрифты, растровые изображения, области, контуры и палитры.

Перо влияет на способ рисования линии. Оно действует как на прямые, так и на кривые линии, а также на границы плоскостных фигур. *Кисть* действует на способ рисования внутренней области плоскостных фигур.

При первичном создании объект контекста устройства содержит заданные по умолчанию перо и кисть. Перо рисует сплошную черную линию шириной в один пиксель, а кисть заливает внутреннюю область сплошным белым цветом.

Инструмент рисования	Инструмент, заданный по умолчанию	Функции рисования, на которые он действует
Перо	BLACK_PEN	Arc, Chord, Ellipse, LineTo, Pie, PolyBezier, PolyBezierTo, Polygon, PolyLine, PolyLineTo, PolyPolygon, PolyPolyLine, Rectangle, RoundRect.
Кисть	WHITE_BRUSH	Chord, Ellipse, ExtFloodFill, FloodFill, Pie, Polygon, PolyPolygon, Rectangle, RoundRect.

Выбор стандартных инструментов рисования

Чтобы изменить текущее перо или кисть, следует выбрать стандартные или создать пользовательские, а затем выбрать их в объекте контекста устройства.

Выбрать перо или кисть можно посредством вызова функции `SelectStockObject` класса `CDC`. Ее параметр `nIndex` является кодом стандартного объекта, возможные значения его перечислены в таблице.

Значение nIndex	Встроенный объект
BLACK_BRUSH	Черная кисть
DKGRAY_BRUSH	Темно-серая кисть
GRAY_BRUSH	Серая кисть
LTGRAY_BRUSH	Светло-серая кисть
NULL_BRUSH	Нулевая кисть (область не закрашивается)
WHITE_BRUSH	Белая кисть (умолчание)
BLACK_PEN	Черное перо (умолчание)
NULL_PEN	Нулевое перо (не рисует линию или границу)
WHITE_PEN	Белое перо

Пример выбора кисти и пера:

```
void CMyView::OnDraw(CDC* pDC)
{
    pDC->SelectStockObject(WHITE_PEN);
    pDC->SelectStockObject(GRAY_BRUSH);
    // Собственно рисование ...
}
```

Создание инструментов рисования

Для создания инструментов рисования необходимо выполнить следующую последовательность действий:

- Создать экземпляр класса CPen (для пера) или CBrush (для кисти).
- Вызвать соответствующую функцию класса CPen или CBrush для инициализации пера или кисти.
- Выбрать перо или кисть в объекте контекста устройства, сохраняя указатель на предыдущее перо или кисть.
- Выполнить собственно рисование, вызывая соответствующие функции.
- Снова выбрать старое перо или кисть в объекте контекста устройства.







Для инициализации пера используется функция CreatePen класса CPen:

```
BOOL CreatePen (int nPenStyle, int nWidth, COLORREF crColor);
```

Параметр nPenStyle задает стиль линии (см. рисунок). Стиль RS_NULL соответствует стандартному перу NULL_PEN. Стиль PS_INSIDEFRAME используется для рисования границы вокруг фигуры с замкнутым контуром, расположенной внутри ограничивающего прямоугольника, и будет рассмотрен далее. Параметр nWidth задает ширину линии в логических единицах. Если он равен нулю, то ширина линии равна одному пикселю, независимо от режима отображения. Параметр crColor задает цвет. Его легче всего описать, используя макрос RGB (примеры возможных значений приведены в следующей таблице):

```
ColorRef RGB (bRed, bGreen, bBlue);
```

Ниже перечислены наиболее часто используемые значения аргументов функций создания пера, задающие его стиль и цвет.

		bRed, bGreen, bBlue	Цвет VGA
PS_SOLID		128, 0, 0	Темно-красный
PS_DASH		255, 0, 0	Светло-красный
PS_DOT		0, 128, 0	Темно-зеленый
PS_DASHDOT		0, 255, 0	Светло-зеленый
PS_DASHDOTDOT		0, 0, 128	Темно-синий
PS_NULL		0, 0, 255	Светло-синий
PS_INSIDEFRAME		128, 128, 0	Темно-желтый
		255, 255, 0	Светло-желтый
		0, 128, 128	Темно-бирюзовый
		0, 255, 255	Светло-бирюзовый
		128, 0, 128	Темно-сиреневый
		255, 0, 255	Светло-сиреневый
		0, 0, 0	Черный
		128, 128, 128	Темно-серый
		192, 192, 192	Светло-серый
		255, 255, 255	Белый

Создание кисти



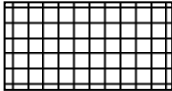

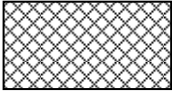
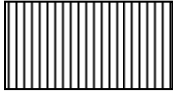
Кисть можно инициализировать так, чтобы она окрашивала однородным цветом внутреннюю область фигур, вызывая функцию `CreateSolidBrush` класса `CBrush`. При этом если цвет не является *чистым* (чистые цвета перечислены в выше приведенной таблице), то Windows имитирует цвет путем смешения.

BOOL CreateSolidBrush (COLORREF crColor);

Инициализировать кисть для заливки внутренней области фигур можно и посредством функции `CreateHatchBrush`:

BOOL CreateHatchBrush (int nIndex, COLORREF crColor);

Параметр `nIndex` задает один из следующих узоров:

HS_BDIAGONAL		HS_FDIAGONAL	
HS_HCROSS		HS_HORIZONTAL	
HS_DIAGCROSS		HS_VERTICAL	

Функцию `CreatePatternBrush` класса `CBrush` инициализирует кисть для заполнения фигуры заданным узором:

```
BOOL CreatePatternBrush (CBitmap* pBitmap);
```

Параметр `pBitmap` является указателем на объект растрового изображения размера 8*8 пикселей. Если изображение монохромное, то Windows использует текущие цвета текста и фона.

Выбор пера или кисти в объекте контекста устройства

Для осуществления этой операции используются следующие функции:

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

Данные функции возвращают указатели на ранее выбранные перо или кисть соответственно.

После вызова графических функций для отображения выводимой информации следует вызвать функцию `SelectObject` для выбора предыдущего объекта, *удалив* тем самым свои перо или кисть из объекта контекста устройства. Это необходимо сделать, чтобы в объекте контекста устройства не хранились некорректные дескрипторы после удаления пера или кисти.

Пример функции OnDraw

```
void CMyView::OnDraw(CDC* pDC)  
{  
    CBrush Brush;  
    CPen Pen;  
    CBrush *PtrOldBrush;  
    CPen *PtrOldPen;  
  
    // Сплошное синее перо шириной 3 пикселя  
    Pen.CreatePen (PS_SOLID, 3, RGB (0, 0, 255));  
    // Сплошная желтая кисть  
    Brush.CreateSolidBrush (RGB (255, 255, 0));  
  
    PtrOldPen = pDC->SelectObject (&Pen);  
    PtrOldBrush = pDC->SelectObject (&Brush);  
  
    // Собственно рисование ...  
  
    // Восстановить перо и кисть  
    pDC->SelectObject (PtrOldPen);  
    pDC->SelectObject (PtrOldBrush);  
}
```

15.3. Установка атрибутов рисования для объекта

Стандартные атрибуты

При первичном создании объект контекста устройства имеет набор стандартных атрибутов, определяющих работу функций рисования. Класс CDC содержит функции для получения значений этих атрибутов и их изменения.

Атрибут рисования	Стандартное значение	Функция для установки или получения значения
Направление рисования дуги	По часовой стрелке	SetArcDirection, GetArcDirection
Цвет фона	Белый	SetBkColor, GetBkColor
Режим фона	OPAQUE	SetBkMode, GetBkMode
Первоначальная кисть	0,0 (координаты экрана)	SetBrushOrg, GetBrushOrg
Текущая позиция	0,0 (логические координаты)	MoveTo, GetCurrentPosition
Режим рисования	R2_COPYPEN	SetROP2, GetROP2
Режим отображения	MM_TEXT	SetMapMode, GetMapMode
Режим заливки	ALTERNATE	SetPolyFillMode, GetPolyFillMode

Режим отображения

Режим отображения (mapping mode) определяет единицы измерения и направление увеличения значений координат, а также воздействует на способ интерпретации координат, передаваемых в функции графического вывода и другие функции, принимающие *логические координаты*.

Координаты устройства в качестве начала координат (точки 0,0) всегда имеют левый верхний угол, а в качестве единиц измерения используют пиксели.

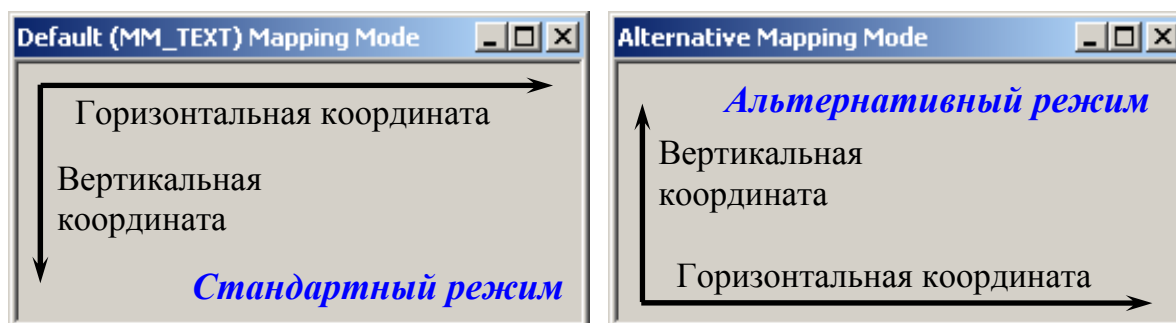
В стандартном режиме отображения (MM_TEXT) логические координаты также используют в качестве единиц измерения пиксели, и направление увеличения координат совпадает с таковым у координат устройства (начало может не совпадать).

Создание альтернативного режима отображения может изменить и логические единицы, и направление увеличения логических координат. Назначить альтернативный режим отображения можно посредством вызова функции SetMapMode класса CDC:

```
virtual int SetMapMode (int nMapMode);
```

Здесь nMapMode – индекс, описывающий новый режим отображения.

Для режимов отображения MM_HIENGLISH, MM_HIMETRIC, MM_LOENGLISH, MM_LOMETRIC и MM_TWIPS горизонтальные координаты увеличиваются вправо, а вертикальные – вверх, в отличие от стандартного режима (MM_TEXT):



В режимах отображения MM_ANISOTROPIC и MM_ISOTROPIC размер логических единиц и направление роста координат настраиваются программистом. При этом вертикальные и горизонтальные координаты в режиме MM_ISOTROPIC имеют одинаковый размер, а в режиме MM_ANISOTROPIC могут быть различными. Для их описания используются функции SetWindowExt и SetViewportExt класса CDC.

Ниже приведена таблица размеров логических единиц для различных режимов отображения.

Значение nMapMode	Размер логической единицы
MM_ANISOTROPIC	Определяется самостоятельно
MM_HIENGLISH	0.001 дюйма
MM_HIMETRIC	0.01 мм
MM_ISOTROPIC	Определяется самостоятельно
MM_LOENGLISH	0.01 дюйма
MM_LOMETRIC	0.1 мм
MM_TEXT	1 единица устройства (пиксель)
MM_TWIPS	1/1440 дюйма (1/20 точки)

Преимуществом использования альтернативных режимов MM_HIENGLISH, MM_HIMETRIC, MM_LOENGLISH, MM_LOMETRIC или MM_TWIPS, а также MM_ANISOTROPIC и MM_ISOTROPIC (при соответствующем задании единиц) – является то, что размер изображения не зависит от устройства, на которое оно выводится, в то время как для стандартного режима (MM_TEXT) размер зависит от разрешения устройства.

15.4. Создание графических изображений

Базовые функции рисования

Класс CDC содержит три базовых функции для того, чтобы узнать или установить цвет пикселя. Функция `SetPixelV` позволяет установить цвет заданного пикселя в конкретное значение. Функция `SetPixel` делает, то же самое, однако, кроме того, возвращает старое значение цвета данного пикселя (и потому работает медленнее). Наконец, функция `GetPixel` позволяет узнать цвет пикселя. Все эти функции позволяют задавать координаты пикселя в двух вариантах: как отдельные координаты, и как структуру POINT:

```
BOOL SetPixelV(int x, int y, COLORREF crColor);
BOOL SetPixelV( POINT point, COLORREF crColor );
COLORREF SetPixel( int x, int y, COLORREF crColor );
COLORREF SetPixel( POINT point, COLORREF crColor );
COLORREF GetPixel( int x, int y ) const;
COLORREF GetPixel( POINT point ) const;
```

Далее в качестве примера использования этих функций для создания изображения будет разработана программа Mandel, выводящая в окно представления множество Мандельброта. При изменении размера или положения окна представления программа производит перерисовку заново.

Замечание. В данной версии программы не следует ее заставлять перерисовывать картинку до того, как закончилось предыдущее рисование.

Программа Mandel – постановка задачи

Пусть $Z = X + iY$ – точка комплексной области.

Рассмотрим последовательность, определяемую итерационной процедурой

$$Z_n = Z_{n+1}^2 + C$$

Прделаем эту процедуру для различных значений C , из области

$$\operatorname{Re} C \in (CR_{\min}; CR_{\max}) \quad \operatorname{Im} C \in (CI_{\min}; CI_{\max})$$

В нашем случае значения C будут браться из области

$$\operatorname{Re} C \in (-2.0; 1.0) \quad \operatorname{Im} C \in (-1.2; 1.2)$$

Если $C = p + iq$, то пересчет вещественной и мнимой составляющей производится по формулам

$$X_{k+1} = X_k^2 - Y_k^2 + p \quad Y_{k+1} = 2X_k Y_k + q$$

Каждому значению C сопоставим цвет, зависящий от того, за сколько итераций точка уходит в "бесконечность" (у нас в программе за границы окружности радиуса 2). Если этого не происходит за достаточно большое число шагов (NMAX), то полагаем, что это не произойдет никогда, и тоже красим точку в заданный цвет.

Программа Mandel – генерация и настройка кода

Сгенерируем исходный код программы посредством мастера AppWizard, повторяя те же шаги и выбирая те же значения параметров, что и для программы WinGreet (за исключением имени проекта – Mandel).

Сначала переопределим виртуальную функцию OnIdle класса приложения. Эта функция вызывается периодически (из главного цикла обработки сообщений MFC) во время простоя программы.

Для этого в окне мастера ClassWizard откроем вкладку Message Maps и выберем класс CMandelApp в списках Class name и Object IDs. Выберем OnIdle в списке Messages и добавим функцию. Отредактируем ее код следующим образом:

```
BOOL CMandelApp::OnIdle(LONG lCount)
{
    // TODO: Добавьте собственный код обработчика
    CWinApp::OnIdle(lCount); // Здесь убрать return!
    CMandelView *PView =
        (CMandelView *) ((CFrameWnd *) m_pMainWnd) ->GetActiveView();
    PView->DrawCol();
    return TRUE;
}
```

Сначала функция вызывает вариант функции OnIdle из базового класса для выполнения стандартных действий программы. Затем вызов CFrameWnd::GetActiveView используется для получения указателя на объект представления. Переменная m_pMainWnd класса приложения сохраняет адрес объекта главного окна. Вызываемая далее функция DrawCol класса представления описана ниже, она рисует один столбец изображения.

Возвращаемое функцией значение TRUE означает, что MFC следует вызывать ее и далее. В противном случае вызовы были бы отменены до получения программой первого сообщения.

Поскольку формирование всего изображения занимает довольно много времени, программа не рисует все изображение из функции OnDraw (а она является обработчиком сообщения). В противном случае на время рисования была бы заблокирована обработка всех других сообщений программы. Поэтому программа рисует только один столбец во время простоя и сразу возвращает управление, чтобы не блокировать обработку сообщений.

Добавим необходимые переменные и константы в класс CMandelView. В файле MandelView.h:


```

class CMandelView : public CView
{
private:
    int m_Col;        // Номер следующего столбца пикселей
    int m_ColMax;     // Последний столбец
    float m_CR;       // Используются
    float m_DCI;      // при масштабировании
    float m_DCR;      // изображения
    int m_RowMax;     // Последняя строка

public:
    void DrawCol (); // Рисование очередного столбца

```

// Оставшаяся часть определения класса

В начале файла реализации класса (MandelView.cpp) добавим следующий код:

```

// ...
#endif

// Набор констант для построения изображения:
#define CIMAX 1.2
#define CIMIN -1.2
#define CRMAX 1.0
#define CRMIN -2.0
#define NMAX 128
// Используемые цвета:
DWORD ColorTable [6] =
    {0x0000ff, // красный
     0x00ff00, // зеленый
     0xff0000, // синий
     0x00ffff, // желтый
     0xffff00, // бирюзовый
     0xff00ff}; // сиреневый
// ...

```

В конструкторе класса CMandelView (файл MandelView.cpp) инициализируем переменную m_Col:

```

CMandelView::CMandelView()
{
    // TODO: Добавьте код конструктора
    m_Col = 0;
}

```

Добавим также определение функции DrawCol (файл MandelView.cpp):

```

void CMandelView::DrawCol ()
{
    CClientDC ClientDC (this);
    float CI;
    int ColorVal;
    float I;
    float ISqr;
    float R;
    int Row;
    float RSqr;

    if (m_Col >= m_ColMax || GetParentFrame ()->IsIconic ())
        return;
    CI = (float)CIMAX;
    for (Row = 0; Row < m_RowMax; ++Row)
    {
        R = (float)0.0;
        I = (float)0.0;
        RSqr = (float)0.0;
        ISqr = (float)0.0;
        ColorVal = 0;
        while (ColorVal < NMAX && RSqr + ISqr < 4)
        {
            ++ColorVal;
            RSqr = R * R;
            ISqr = I * I;
            I *= R;
            I += I + CI;
            R = RSqr - ISqr + m_CR;
        }
        ClientDC.SetPixelV (m_Col, Row, ColorTable[ColorVal%6]);
        CI -= m_DCI;
    }
    m_Col++;
    m_CR += m_DCR;
}

```

Функция DrawCol рисует очередной столбец пикселей, продвигаясь слева направо. Для закраски пикселя используется функция SetPixelV. Функция DrawCol сразу возвращает управление, если главное окно минимизировано (вызов CWnd::IsIconic возвращает TRUE). Адрес объекта главного окна для вызова IsIconic получается вызовом функции GetParentFrame.

Теперь добавим обработчик сообщения WM_SIZE. В окне мастера ClassWizard откроем вкладку Message Maps и выберем класс CMandelView в списках Class name и Object IDs. Выберем WM_SIZE в списке Messages и

добавим обработчик (имя по умолчанию – OnSize). Отредактируем ее код следующим образом:

```
void CMandelView::OnSize(UINT nType, int cx, int cy)
{
    CView::OnSize(nType, cx, cy);
    // TODO: Добавьте собственный код обработчика
    if (cx <= 1 || cy <= 1) // предотвратим деление на ноль
        return;
    m_ColMax = cx; // Текущие размеры окна представления
    m_RowMax = cy; // в единицах устройства (пикселях)
    m_DCR = (float)((CRMAX - CRMIN) / (m_ColMax-1));
    m_DCI = (float)((CIMAX - CIMIN) / (m_RowMax-1));
}
```

После вызова функции OnSize окно представления очищается, а для перерисовки вызывается функция OnDraw. Она же вызывается и при необходимости перерисовки по любой другой причине. Наш вариант функции OnDraw не рисует ничего – он просто устанавливает значение текущего столбца в ноль, чтобы функция OnIdle начала рисование сначала. Код функции OnDraw:

```
void CMandelView::OnDraw(CDC* pDC)
{
    CMandelDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    // TODO: Добавьте код отображения собственных данных
    m_Col = 0;
    m_CR = (float)CRMIN;
}
```

В завершение работы над программой добавим в функцию InitInstance (файл Mandel.cpp) вызов функции SetWindowText для формирования заголовка программы:

```
// ...
m_pMainWnd->UpdateWindow();
m_pMainWnd->SetWindowText ("Mandelbrot Demo");
return TRUE;
}
```

15.5. Функции рисования - члены класса CDC

Прямые линии

Ниже приведены прототипы основных функций, используемых для рисования прямых линий:

```

// Сдвинуть текущую графическую позицию.
// Возвращается прежняя позиция.
CPoint MoveTo( int x, int y );
CPoint MoveTo( POINT point );

// Провести линию в точку. Возвращает TRUE, если нарисована
BOOL LineTo( int x, int y );
BOOL LineTo( POINT point );

// Провести ломаную. Возвращает TRUE в случае успеха
BOOL Polyline( LPPOINT lpPoints, int nCount );

```

Первый аргумент функции **Polyline** задает указатель на массив из элементов типа **POINT**, второй – количество точек. В отличие от первых двух функций **Polyline** не изменяет значения текущей графической позиции.

Регулярные кривые – дуга

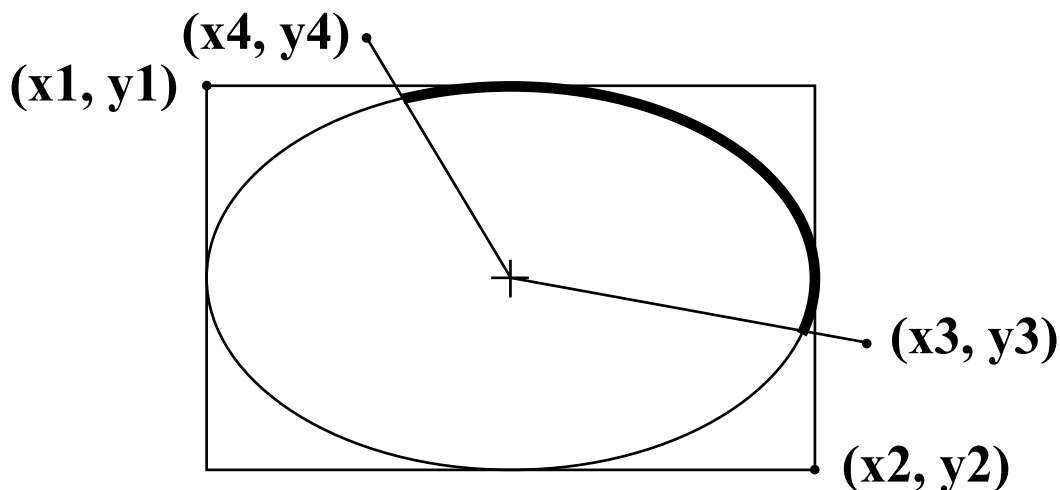
Сегменты эллипсов рисуют с помощью функции **CDC::Arc**:

```

BOOL Arc
( int x1, int y1,    // левый верхний угол прямоугольника
  int x2, int y2,    // правый нижний угол прямоугольника
  int x3, int y3,    // начальная точка дуги
  int x4, int y4 ); // конечная точка дуги

// Другой вариант
BOOL Arc( LPCRECT lpRect, POINT ptStart, POINT ptEnd );

```



Регулярные кривые – кривая Безье

Кривые Безье (третьего порядка) рисуют с помощью функции **CDC::PolyBezier**:

```

BOOL PolyBezier
( const POINT* lpPoints, // указатель на массив точек
  int nCount );          // количество точек

```

Ниже приводится фрагмент программы, рисующей кривую Безье, состоящую из трех фрагментов (следовательно, требуется 10 точек). Описание массива:

```
POINT m_apt[10];
```

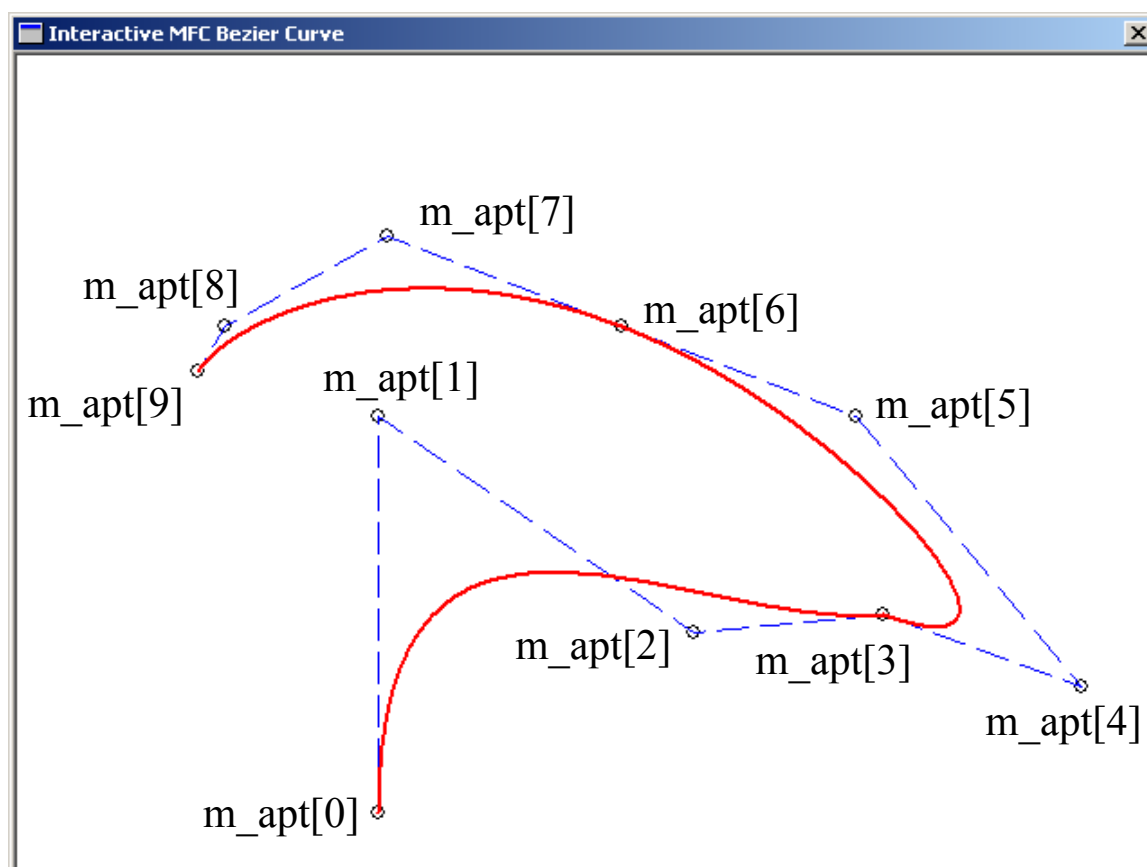
Заполнение массива:

```
m_apt[0].x = 200; m_apt[0].y = 420;
m_apt[1].x = 200; m_apt[1].y = 200;
m_apt[2].x = 375; m_apt[2].y = 320;
m_apt[3].x = 480; m_apt[3].y = 310;
m_apt[4].x = 590; m_apt[4].y = 350;
m_apt[5].x = 465; m_apt[5].y = 200;
m_apt[6].x = 335; m_apt[6].y = 150;
m_apt[7].x = 205; m_apt[7].y = 100;
m_apt[8].x = 115; m_apt[8].y = 150;
m_apt[9].x = 100; m_apt[9].y = 175;
```

Собственно рисование:

```
void CMainWnd::DoBezier()
{
    CClientDC dc(this);
    for (int i = 0; i < 10; i++) // Нарисовать контрольные точки
        dc.Ellipse(m_apt[i].x - 4, m_apt[i].y - 4,
                   m_apt[i].x + 4, m_apt[i].y + 4);
    // Создать новые перья
    CPen penBlue, penRed;
    penBlue.CreatePen(PS_DASH, 1, crBlue);
    penRed.CreatePen(PS_SOLID, 2, crRed);
    // Выбрать новое перо в контекст устройства, сохранив старое
    CPen* ppenOld = dc.SelectObject(&penBlue);
    // Соединить контрольные точки синими линиями
    dc.Polyline(m_apt, 10);
    // Нарисовать кривую Безье красным пером
    dc.SelectObject(&penRed);
    dc.PolyBezier(m_apt, 10);
    // Восстановить прежнее перо
    dc.SelectObject(ppenOld);
}
```

Результат представлен на рисунке (синие линии изображены пунктиром). Обратите внимание, что в точке 6 кривая гладкая, а в точке 3 – нет.



Режим рисования линий, режим фона

Кроме стиля, толщины и цвета линий, определяемых установками пера, результат рисования зависит также от текущего *режима рисования*, который описывает способ комбинирования цвета пера с текущим цветом дисплея. Изменить режим можно вызовом функции **SetRop2** класса CDC:

```
int SetRop2 (int nDrawMode);
```

Наиболее часто используемые режимы (из 16 возможных) перечислены в следующей таблице:

Значение nDrawMode	Цвет каждого пикселя рисуемой линии
RC_COPYPEN (по умолчанию)	Цвет пера
RC_NOTCOPYPEN	Инверсный цвету пера
R2_NOT	Инверсный цвету фона
R2_BLACK	Черный
R2_WHITE	Белый
R2_NOP	Не изменяется

При рисовании прерывистых линий цвет, используемый для закрашивания промежутков, зависит от текущего режима фона (`CDC::SetBkMode`) и его цвета (`CDC::SetBkColor`). Если задан режим `OPAQUE`, пропуски закрашиваются цветом фона, если `TRANSPARENT` – не закрашиваются вовсе.

Рисование замкнутых линий

Функции класса `CDC`, позволяющие рисовать фигуры с замкнутым контуром:

- `Rectangle` – прямоугольник
- `RoundRect` – закругленный прямоугольник
- `Ellipse` – эллипс
- `Chord` – сегмент
- `Pie` – сектор
- `Polygon` – многоугольник

Далее приводятся прототипы функций. Имеются также альтернативные прототипы, где вместо отдельных координат в качестве аргументов используются структуры, задающие точки и/или прямоугольники.

`BOOL Rectangle`

```
( int x1, int y1,      // левый верхний угол прямоугольника
  int x2, int y2 );    // правый нижний угол прямоугольника
```

`BOOL RoundRect`

```
( int x1, int y1,      // левый верхний угол прямоугольника
  int x2, int y2,      // правый нижний угол прямоугольника
  int x3, int y3 );    // размер закругляющих эллипсов
```

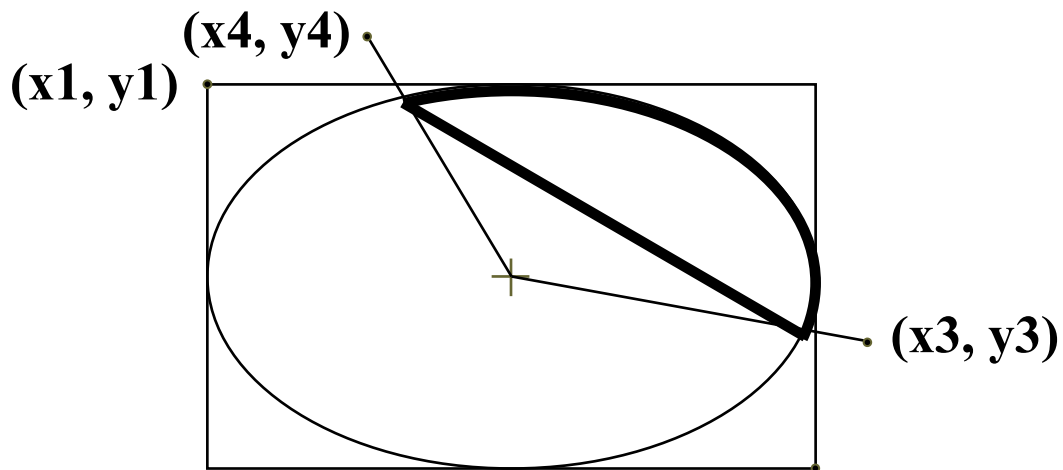
`BOOL Ellipse`

```
( int x1, int y1,      // левый верхний угол
                          // ограничивающего прямоугольника
  int x2, int y2 );    // правый нижний угол
                          // ограничивающего прямоугольника
```

`BOOL Chord`

```
( int x1, int y1,      // левый верхний угол
                          // ограничивающего прямоугольника
  int x2, int y2,      // правый нижний угол
                          // ограничивающего прямоугольника
  int x3, int y3,      // начальная точка хорды
  int x4, int y4 );    // конечная точка хорды
```

Результат вызова функции Chord представлен на следующем рисунке.

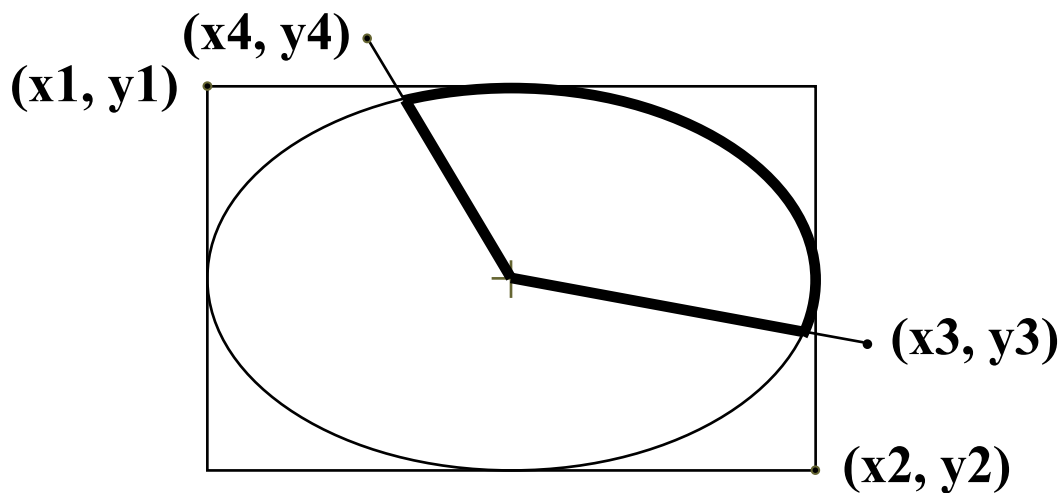


```

BOOL Pie
( int x1, int y1,      // левый верхний угол
  int x2, int y2,      // ограничивающего прямоугольника
  int x3, int y3,      // правый нижний угол
  int x4, int y4 );    // ограничивающего прямоугольника
                        // начальная точка сектора
                        // конечная точка сектора

```

Результат вызова функции Pie представлен на рисунке.



```

BOOL Polygon
( LPPOINT lpPoints,    // массив точек (вершины)
  int nCount );        // их количество

```

Другие функции рисования

В таблице перечислено еще несколько часто используемых функций рисования из класса CDC.

Функция	Назначение
DrawFocusRect	Рисует границу прямоугольника пунктирной линией цветом, инверсным цвету экрана, без заливки внутренней области. Повторное рисование с теми же параметрами удаляет границу.
DrawIcon	Рисует значок.
ExtFloodFill	Заполняет область, ограниченную заданным цветом, используя текущую кисть. Можно закрасить в том числе область, <i>уже заполненную</i> цветом границы.
FillRect	Заполняет прямоугольную область, используя указанную кисть, без рисования границ.
FloodFill	Заполняет область, ограниченную заданным цветом, используя текущую кисть.
FrameRect	Рисует прямоугольную границу, используя указанную кисть, без заполнения внутренней области.
InvertRect	Инвертирует цвет внутри прямоугольной области.
PolyDraw	Рисует фигуры, состоящие из комбинаций прямых и кривых линий (сегментов прямых и кривых Безье).

15.6. Пример – программа *MiniDraw*

В качестве примера использования рассмотренных средств мы приведем очередную версию программы *MiniDraw*. Поскольку полное описание довольно объемно, мы ограничимся рассмотрением основных изменений по сравнению с ранее разработанной версией (в теме "Перемещаемые панели и строки состояния").

Изменения в интерфейсе

Вместо пункта главного меню *Lines* введен пункт *Options* с двумя подпунктами *Color* и *Line Thickness* (оба *Pop-up*). Первый служит для вызова всплывающего меню для выбора цвета, второй – такого же меню для выбора толщины линии. Читателям предлагается самостоятельно модифицировать меню программы, а также создать функции-обработчики (определившись с тем, какому классу они должны принадлежать). Функции обработчики должны быть созданы как для обработки сообщения *COMMAND*, так и для сообщения *UPDATE_COMMAND_UI*.

Также следует завести члены-переменные в этом классе для запоминания текущего цвета и идентификатора соответствующего пункта всплывающего меню *Color* (для толщины линии это уже было сделано).

Примерный код функций-обработчиков приведен ниже.

```

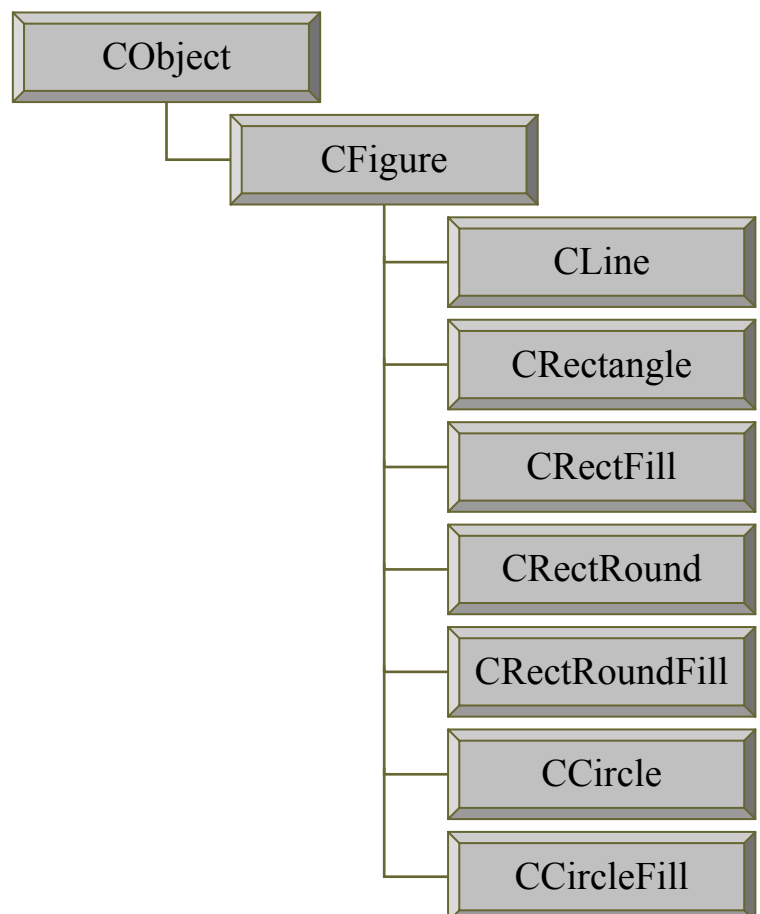
void CMiniDrawApp::OnColorBlack()
{
    m_CurrentColor = RGB (0,0,0);
    m_IdxCmd = ID_COLOR_BLACK;
}
void CMiniDrawApp::OnUpdateColorBlack(CCmdUI* pCmdUI)
{
    pCmdUI->SetCheck (m_IdxCmd==ID_COLOR_BLACK ? 1 : 0);
}
void CMiniDrawApp::OnColorCustom()
{
    CColorDialog ColorDialog; // Диалог выбора цвета (станд.)
    if (ColorDialog.DoModal () == IDOK)
    {
        m_CurrentColor = ColorDialog.GetColor ();
        m_IdxCmd = ID_COLOR_CUSTOM;
    }
}
void CMiniDrawApp::OnUpdateColorCustom(CCmdUI* pCmdUI)
{
    pCmdUI->SetCheck (m_IdxCmd==ID_COLOR_CUSTOM ? 1 : 0);
}

```

Определение классов для фигур

В предыдущей версии программы, рисовавшей только линии, был определен класс CLine для хранения информации о единственном используемом типе изображаемой фигуры. Поскольку мы намереваемся организовать хранение в памяти, запись и чтение из файла и рисование разных типов объектов, разумно выстроить иерархию классов, как это показано на диаграмме.

Соответственно, в файле MiniDrawDoc.h вместо описания класса CLine появится описание всех классов, отображенных на диаграмме справа. Фрагменты этого кода приведены ниже.



```

class CFigure : public CObject
{
protected:
    COLORREF m_Color;
    DWORD m_X1, m_Y1, m_X2, m_Y2;
    CFigure () {}
    DECLARE_SERIAL (CFigure)
public:
    virtual void Draw (CDC *PDC) {}
    CRect GetDimRect ();
    virtual void Serialize (CArchive& ar);
};
class CLine : public CFigure
{
protected:
    DWORD m_Thickness;
    CLine () {}
    DECLARE_SERIAL (CLine)
public:
    CLine (int X1, int Y1, int X2, int Y2,
COLORREF Color, int Thickness);
    virtual void Draw (CDC *PDC);
    virtual void Serialize (CArchive& ar);
};

```

Определение классов для фигур – комментарии

Для порождаемых классов фигур у нас используется базовый класс CFigure, порожденный в свою очередь от CObject. Класс CFigure содержит переменные и функции, используемые для *всех* фигур, а порожденные от него классы – переменные и функции, используемые для фигуры конкретного типа.

Так, класс CFigure содержит переменные для хранения координат ограничивающего прямоугольника и цвета всех фигур. А переменную для хранения толщины линии содержат только классы, соответствующие линиям и незакрашенным фигурам. CFigure предоставляет функцию для получения ограничивающего прямоугольника (GetDimRect).

Класс CFigure также содержит виртуальную функцию Serialize для чтения с диска и записи на диск информации о фигуре. Производные классы, содержащие дополнительные данные, должны иметь собственную функцию сериализации для записи/чтения этих данных.

Замечание. Классы, использующие сериализацию, в своем файле определения должны содержать макрос DECLARE_SERIAL, а в файле реализации – макрос IMPLEMENT_SERIAL.

Класс CFigure содержит виртуальную функцию Draw. Наличие этого определения – а она определена как пустая – позволяет программе использовать единственный указатель класса CFigure для вызова функции Draw применительно к фигуре любого типа. Реализацию данных функций студентам предлагается изучить самостоятельно по предложенному им исходному коду.

Другие модификации программы

Для класса CMiniDrawDoc:

Вместо массива m_LineArray и функций AddLine, GetLine, GetNumLines введен массив m_FigArray и функции AddFigure, GetFigure, GetNumFig соответственно. Соответствующий код достаточно очевиден.

Для класса CMiniDrawView:

В описании добавлена переменная m_PenDotted для пера, используемого для рисования пунктирной рамки в процессе задания положения фигуры мышью (dragging) в функции OnMouseMove. Для изменения вида этой временной фигуры она сначала прорисовывается повторно для уничтожения старого изображения (используется режим R2_NOT), а затем выводится уже с новым положением (или размерами) – смотри исходный код.

Изменен код функции OnLButtonUp. После определения типа фигуры (по значению переменной m_CurrentTool класса приложения) производится окончательная прорисовка фигуры, затем создается соответствующий объект и добавляется в массив m_FigArray класса документа.

В коде функции OnDraw изменено несколько строк, связанных с изменениями в составе функций класса CMiniDrawDoc.

Более детально изучить работу программы студентам предлагается самостоятельно по предложенному им исходному коду.

Тема 16. Растровые изображения и битовые операции

16.1. Создание растровых изображений

Библиотека MFC для управления растровыми изображениями содержит класс `CBitmap`. Поэтому при их создании первым делом следует создать экземпляр этого класса. Обычно он объявляется как переменная-член одного из главных классов программы, например, класса представления.

После этого необходимо вызвать соответствующие функции этого класса для инициализации объекта. Мы подробно рассмотрим два варианта, а именно:

- вызов функции `LoadBitmap` для загрузки изображения из ресурсов программы;
- вызов функции `CreateCompatibleBitmap` для создания пустого растрового изображения для его формирования при выполнении программы.
- Кроме этих двух методов имеется также ряд других, в частности:
- функция `CBitmap::LoadOEMBitmap` для загрузки предопределенного растрового изображения, предоставляемого Windows;
- функция `CBitmap::CreateBitmap` для создания растрового изображения с заданными параметрами для вывода на конкретное устройство;
- функция `CGdiObject::Attach` для инициализации объекта дескриптора растрового изображения Windows. Класс `CGdiObject` является базовым для класса `CBitmap`.

Загрузка растрового изображения из ресурсов

Для создания и редактирования изображений с количеством цветов не более 256 можно воспользоваться встроенным графическим редактором среды Developer Studio. Для этого можно, например, вызвать редактор ресурсов (`Ctrl+R`) и выбрать тип ресурса `Bitmap`.

Растровое изображение, разработанное с помощью любого графического редактора, можно также включить в ресурсы программы и затем загрузить посредством функции `LoadBitmap`. Данный метод разумно применять для создания относительно сложных рисунков. Соответствующий код выглядит примерно так:

```
// Определение класса представления
class CProgView : public CView
{
// ...
    CBitmap m_Bitmap;
    void LoadBitmapImage();
// ...
};
```

```
// Реализация класса представления
class CProgView::LoadBitmapImage()
{
// ...
    m_Bitmap.LoadBitmap(IDB_BITMAP1);
// ...
}
```

Создание растрового изображения с использованием функций рисования

Альтернативным способом создания рисунка является инициализация пустого растрового изображения и использование функций рисования MFC. Для этого необходимо выполнить следующие действия:

- Инициализировать пустое растровое изображение.
- Создать объект памяти контекста устройства.
- Выбрать растровое изображение в объекте памяти контекста устройства.
- Нарисовать требуемое изображение, вызывая функции класса CDC для объекта памяти контекста устройства.

Для инициализации пустого растрового изображения используется функция `CreateCompatibleBitmap`. Например:

```
// Определение класса представления
class CProgView : public CView
{
// ...
    CBitmap m_Bitmap;
    void DrawBitmapImage();
// ...
};

// Реализация класса представления
class CProgView::DrawBitmapImage()
{
    // Создание объекта контекста устройства окна представления
    CClientDC ClientDC (this);
    m_Bitmap.CreateCompatibleBitmap(&ClientDC, 32, 32);
// ...
}
```

Аргументами функции `CreateCompatibleBitmap` являются адрес объекта контекста устройства, а также ширина и высота изображения (в пикселях).

Создаваемое изображение *совместимо* с устройством, соответствующим объекту. Этот термин означает, что информация об изображении будет структурирована способом, подобным способу структурирования данных самим устройством, и, следовательно, будет передаваться быстро.

Прежде, чем вызывать функции рисования, необходимо создать объект контекста устройства, связанный с изображением, созданным функцией `CreateCompatibleBitmap`. Этот объект называется *объектом памяти контекста устройства*. Для его создания требуется сначала объявить экземпляр класса `CDC`, а затем вызвать функцию этого класса:

```
// ...
CDC MemDC; // Объект памяти контекста устройства
m_Bitmap.CreateCompatibleBitmap(&ClientDC, 32, 32);
MemDC.CreateCompatibleDC(&ClientDC);
// ...
```

В результате этих действий растровое изображение и объект памяти контекста устройства, используемый для доступа к изображению, оказываются совместимы с одним устройством (в нашем случае с экраном).

Замечание. При вызове функции `CreateCompatibleDC` с аргументом `NULL` соответствующий объект памяти контекста устройства инициализируется.

Затем следует вызвать функцию `SelectObject` класса `CDC`, чтобы выбрать объект растрового изображения в объекте памяти контекста устройства. И только после этого можно вызывать функции рисования класса `CDC`.

Далее приводится пример кода, осуществляющего все описанные действия.

Пример создания растрового изображения с использованием функций рисования

```
// Реализация класса представления
class CProgView::DrawBitmapImage()
{
    // Создание объекта контекста устройства окна представления
    CClientDC ClientDC (this);
    CDC MemDC; // Объект памяти контекста устройства
    // Инициализация пустого растрового изображения
    m_Bitmap.CreateCompatibleBitmap (&ClientDC, 32, 32);
    // Инициализация объекта памяти контекста устройства (КУ)
    MemDC.CreateCompatibleDC (&ClientDC);
```

```

// Передача объекта изображения в объект памяти КУ
MemDC.SelectObject (&m_Bitmap);
// Рисование белого фона
MemDC.PatBlt (0, 0, 32, 32, WHITENESS);
// Рисование круга
MemDC.Ellipse (2, 2, 30, 30);
// Вызов других функций рисования ...
}

```

Отображение растрового изображения

Библиотека MFC и средства Win32 API не содержат функции, которую можно было бы просто вызвать для отображения растрового изображения на устройстве. Поэтому мы создадим для этой цели свою функцию:

```

void DisplayBitmap (CDC *PDC, CBitmap *PBitmap, int X, int Y)
{
    BITMAP BM; // Структура для описания изображения
    CDC MemDC; // Объект памяти контекста устройства
    MemDC.CreateCompatibleDC (NULL);
    MemDC.SelectObject (PBitmap);
    PBitmap->GetObject (sizeof (BM), &BM);
    PDC->BitBlt
    (X, // Логическая гориз. координата приемника
    Y, // Логическая верт. координата приемника
    BM.bmWidth, // Ширина перемещаемого блока (в лог. ед.)
    BM.bmHeight, // Высота перемещаемого блока (в лог. ед.)
    &MemDC, // КУ источника для графических данных
    0, // Лог. гориз. координата блока внутри источника
    0, // Лог. верт. координата блока внутри источника
    SRCCOPY); // Код типа перемещения
}

```

Первый параметр приведенной выше функции содержит адрес объекта контекста устройства для вывода растрового изображения, адрес которого задается вторым параметром. Этот объект растрового изображения необходимо инициализировать, используя один из способов (*совместимых с экраном*), описанных ниже. Последние два параметра задают горизонтальную и вертикальную координату позиции внутри целевого устройства, соответствующей верхнему левому углу изображения.

Сначала функция создает объект контекста устройства, совместимый с экраном, и передает изображение внутрь этого объекта, поэтому он может теперь иметь доступ к содержимому изображения. Затем вызов функции `GetObject` класса `CGdiObject` заполняет поля структуры `BITMAP` информацией о изображении.

Вызов функции `BitBlt` класса `CDC` перемещает графические данные, содержащиеся в растровом изображении прямо на целевое устройство. Значения ее параметров достаточно очевидны. Поясним только, что пятый и шестой аргумент имеют значение 0, так как мы передаем изображение целиком, а значение `SRCCOPY` показывает, что оно копируется без изменений.

Функцию `DisplayBitmap` можно использовать для вывода как на экран, так и на принтер. Пример ее вызова:

```
class CProgView::OnDraw (CDC* pDC)
{
    DisplayBitmap (pDC, &m_Bitmap, 0, 0);
}
```

Другие способы использования растровых изображений

Созданный и инициализированный объект `CBitmap` в программах MFC можно использовать и в иных целях (кроме отображения на соответствующем устройстве вывода).

В качестве примера укажем несколько возможных применений:

- С помощью MFC-класса `CBitmapButton` можно создать элемент управления типа кнопки, помеченной растровым изображением, а не текстом.
- Вызывая функцию `CMenu::SetMenuItemBitmap`, можно пометить команду меню специальным значком. Временный объект класса `CMenu`, подключаемый к меню главной программы, можно получить, вызывая функцию `CWnd::GetMenu` для объекта главного окна.
- Вызывая функцию `CMenu::AppendMenu` или другую функцию класса `CMenu`, можно сконструировать специальную метку меню и задать растровое изображение вместо текстовой метки.
- Вызывая функцию `CBrush::CreatePatternBrush` для создания кисти, можно заполнять области с помощью специального шаблона.

16.2. Выполнение битовых операций при отображении

Класс `CDC` содержит три универсальных и эффективных функции для перемещения блоков графических данных: `PatBlt`, `BitBlt` и `StretchBlt`. Они позволяют также осуществлять различные модификации изображений, такие как инвертирование цветов, зеркальное отображение и т.п.

Эти функции можно применять для копирования графических данных на отображающей поверхности одного устройства или для копирования их с одного устройства на другое, а также для обмена данными между устройством и растровым изображением.

Функция *PatBlt*

Функция *PatBlt* класса *CDC* предназначена для закрашивания прямоугольной области с использованием текущей кисти. В этом контексте обычно используют слово *шаблон* (*pattern*). По сравнению с функцией *CDC::FillRect* она более универсальна. Ее синтаксис:

```
BOOL PatBlt
( int x, int y,           // Логические координаты левого
                             // верхнего угла области
  int nWidth, int nHeight, // Размеры области заполнения
                             // в логических единицах
  DWORD dwRop);           // Код растровой операции
```

Последний параметр *dwRop* – это код *растровой операции*. Он задает способ объединения пикселя приемника и пикселя шаблона (т.е. текущей кисти в объекте контекста устройства *приемника*) для получения окончательного значения пикселя, определяющего цвет. В следующей таблице перечислены возможные значения этого параметра. В ней принято, что *D* задает пиксель приемника, а *P* – пиксель шаблона. Операция производится над каждым битом изображения. Результат зависит от количества цветов, которое может отображать устройство.

Код растровой операции	Булево выражение	Описание результата в области приемника
BLACKNESS	$D = 0$	Каждый пиксель устанавливается черным
DSTINVERT	$D = \sim D$	Цвет каждого пикселя инвертируется
PATCOPY	$D = P$	Каждому пикселю задается цвет пикселя шаблона
PATINVERT	$D = D \wedge P$	Цвет каждого пикселя есть результат логической операции XOR над пикселем приемника и пикселем шаблона
WHITENESS	$D = 1$	Каждый пиксель устанавливается белым

Функция *BitBlt*

Функция *BitBlt* класса *CDC* передает блок графических данных из одного места хранения в другое. Источник и приемник могут находиться внутри одного устройства (растрового изображения) или разных. Синтаксис функции:

```

BOOL BitBlt
( int x, int y,                // Логические координаты левого
                                // верхнего угла области
  int nWidth, int nHeight,    // Размеры области заполнения
                                // в логических единицах
  CDC* pSrcDC,                // Объект контекста устройства
                                // источника
  int xSrc, int ySrc,         // Логические координаты левого
                                // верхнего угла блока внутри
                                // источника
  DWORD dwRop);               // Код растровой операции

```

Пример вызова функции `BitBlt` был приведен выше в коде функции для отображения растрового рисунка `DisplayBitmap`.

В отличие от функции `PatBlt` параметр `dwRop` определяет код растровой операции, производимой для получения окончательного значения пикселя, определяющего цвет, с участием трех аргументов: пикселя приемника, пикселя источника и пикселя шаблона. Теоретически он может принимать 256 различных значений.

В таблице перечислены наиболее распространенные значения этого параметра, для которых определены соответствующие константы (в файле `wingdi.h`). Возможно также использование значений, перечисленных в таблице для функции `PatBlt`.

В таблице принято, что *D* задает пиксель приемника, *S* – пиксель источника, а *P* – пиксель шаблона. Операция производится над каждым битом изображения. Результат зависит от количества цветов, которое может отображать устройство.

Код растровой операции	Булево выражение	Описание результата в области приемника
MERGECOPY	$D = P \& S$	Побитовая операция AND над пикселями шаблона и источника
MERGEPAINT	$D = \sim P \mid S$	Побитовая операция OR над пикселем источника и инвертированным пикселем шаблона
NOTSRCCOPY	$D = \sim S$	Итоговый цвет получается инвертированием пикселя источника
NOTSRCERASE	$D = \sim(D \mid S)$	Побитовая операция OR над пикселями приемника и источника с последующим инвертированием результата

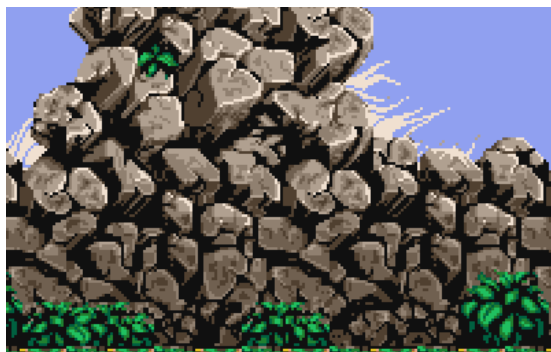
Код растровой операции	Булево выражение	Описание результата в области приемника
PATPAINT	$D = \sim D \mid S \mid P$	Побитовая операция OR над инвертированным пикселем приемника и пикселями источника и шаблона
SRCAND	$D = D \& S$	Побитовая операция AND над пикселями приемника и источника
SRCCOPY	$D = S$	Копирование пикселя источника
SRCERASE	$D = \sim D \mid S$	Побитовая операция OR над инвертированным пикселем приемника и пикселем источника
SRCINVERT	$D = D \wedge S$	Побитовая операция XOR над пикселями приемника и источника
SRCAND	$D = D \mid S$	Побитовая операция OR над пикселями приемника и источника

Использование функции *BitBlt* для анимации

При написании игр и ряда других приложений часто бывает необходимо перемещать некоторый объект на сложном фоне (клавиатурой, мышью или автоматически по таймеру). Для того, чтобы наложить изображение объекта на фон, необходимо для каждой фазы движения заготовить не одно, а два изображения: собственно рисунок на черном фоне и его черную маску на белом фоне. Наложение рисунка на фоновое изображение осуществляется в два этапа: сначала с использованием операции SRCAND накладывается маска, затем с помощью операции SRCINVERT накладывается собственно рисунок объекта.

```
class CProgView::DisplayDrawing (int X, int Y)
{
    CClientDC ClientDC (this);
    CDC MemDC;
    MemDC.CreateCompatibleDC (&ClientDC);
    MemDC.SelectObject (&m_MaskBitmap);    // Маска
    PDC->BitBlt
        (X, Y, BMWIDTH, BMHEIGHT, &MemDC, 0, 0, SRCAND);
    MemDC.SelectObject (&m_ImageBitmap);    // Изображение
    PDC->BitBlt
        (X, Y, BMWIDTH, BMHEIGHT, &MemDC, 0, 0, SRCINVERT);
}
```

Если объект должен двигаться, то дополнительно следует обеспечить сохранение и восстановление прямоугольного участка фонового изображения (SRCCOPY).

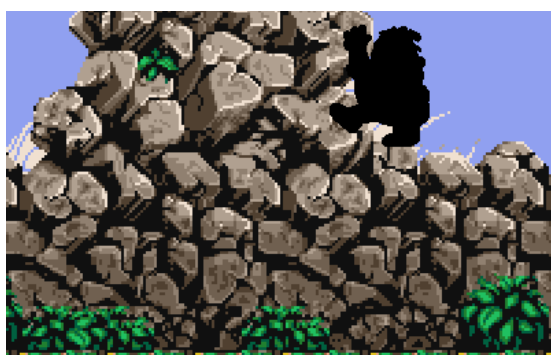


Маска



Изображение

Фоновое изображение



Наложение маски
(операция SRCAND)



Наложение изображения
(операция SRCINVERT)

Функция StretchBlt

Функция StretchBlt класса CDC является самой универсальной из всех перечисленных. Она допускает выполнение всех операций, возможных при использовании функции BitBlt, и кроме того, позволяет *изменить размер* блока графических данных или *повернуть блок*. Синтаксис функции:

```

BOOL StretchBlt
( int x, int y,                // Логические координаты левого
  int nWidth, int nHeight,     // верхнего угла области
  CDC* pSrcDC,                // Размеры области заполнения
  int xSrc, int ySrc,         // в логических единицах
  int nSrcWidth,              // Объект контекста устройства
  int nSrcHeight,             // источника
  DWORD dwRop);               // Логические координаты левого
                                // верхнего угла блока внутри
                                // источника
                                // Ширина блока источника и
                                // высота блока источника
                                // в логических единицах
                                // Код растровой операции

```

Функция `StretchBlt` получает размер блока источника и размер блока приемника. Если размер приемника (`nWidth`, `nHeight`) меньше размера источника (`nSrcWidth`, `nSrcHeight`), то изображение сжимается, если наоборот – растягивается. Если значения `nWidth` и `nSrcWidth` заданы с противоположными знаками, то изображение зеркально отобразится по горизонтали, если противоположны знаки `nHeight` и `nSrcHeight` – по вертикали.

Пример использования функции `StretchBlt` приводится в демонстрационной программе, рассматриваемой далее. В ней она используется для заполнения данным изображением всего окна представления.

Замечание. При сжатии блока изображения возможны разные варианты удаления пикселей (с использованием различных битовых операций) в зависимости от установленного режима. Для того, чтобы узнать или установить режим обработки изображения, можно воспользоваться функциями `GetStretchBltMode` или `SetStretchBltMode` соответственно. Более подробно об этом можно узнать, обратившись к документации.

16.3. Отображение значков

Значок – это специальная форма растрового изображения. Он отличается от прочих изображений двумя признаками.

Во-первых, он может содержать несколько изображений различных размеров (обычно predetermined) и с разным количеством цветов. Это позволяет системе при отображении значка выбрать изображение, наиболее подходящее текущему видеорежиму.

Во-вторых, при использовании специальных редакторов для разработки значка (в том числе встроенный в Visual C++ графический редактор) можно использовать два специальных цвета: *цвет экрана* и *инверсный цвет экрана*. Первый при наложении значка на какой-либо фон будет прозрачным, а второй инвертирует цвет фона, и следовательно, будет виден в любом случае.

Ранее мы видели, что значок можно отобразить

- в строке заголовка главного окна программы или дочернего окна MDI-приложения;
- внутри диалогового окна.

Рассмотрим способ отображения значков в любом месте внутри окна программы.

Прежде всего нужно сконструировать значок, например, во встроенном графическом редакторе Visual C++. Можно также импортировать его из файла `.ico`, полученного другим способом.

Если значку присвоить идентификатор `IDR_MAINFRAME` (или идентификатор типа документа программы в MDI-приложении, в рассмотренном ранее примере – `IDR_TEXTTYPE`), то он будет автоматически назначаться главному или дочернему окну и отображаться в строке заголовка.

Если это не требуется, то следует выбрать другой идентификатор (Visual C++ предлагает имена `IDI_ICON1`, `IDI_ICON2` и т.д.).

Созданный или импортированный значок включается в программные ресурсы при построении программы. Перед его отображением его необходимо загрузить из ресурсов, используя функцию `LoadIcon`:

```
HICON hIcon;  
hIcon = AfxGetApp() -> LoadIcon(IDI_ICON1);
```

Здесь функция `AfxGetApp` используется для получения указателя на объект приложения. Если ресурс содержит несколько изображений, то автоматически будет загружено то из них, которое наиболее подходит для текущего видеорежима.

Для вызова предопределенных значков, предоставляемых Windows, можно воспользоваться функцией `LoadStandardIcon` или `LoadOEMIcon`.

Для отображения значка используется функция `DrawIcon` класса `CDC`:

```
BOOL DrawIcon (int x, int y, HICON hIcon);
```

Ее аргументы задают координаты левого верхнего угла места расположения значка, а также дескриптор, получаемый через вызов функций `LoadIcon`, `LoadStandardIcon` или `LoadOEMIcon`.

Пример функции, которая загружает и отображает в центре окна представления значок, созданный или импортированный в проект:

```
class CProgView::DisplayIcon ()  
{  
    CClientDC ClientDC (this);  
    HICON hIcon;  
    int IconHeight, IconWidth;  
    RECT Rect;  
    hIcon = AfxGetApp() -> LoadIcon (IDI_ICON1);  
    GetClientRect (&Rect);  
    IconWidth = GetSystemMetrics (SM_CXICON);  
    IconHeight = GetSystemMetrics (SM_CYICON);  
    ClientDC.DrawIcon (Rect.right/2 - IconWidth/2,  
                      Rect.bottom/2 - IconHeight/2, hIcon);  
}
```

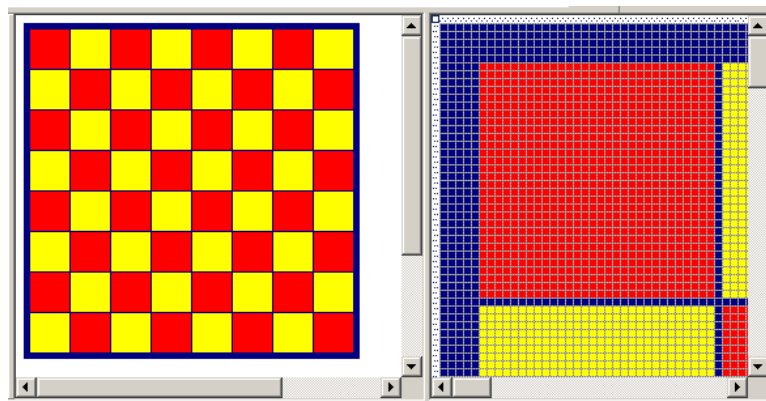
В этом примере функция `GetSystemMetrics` используется для получения размера значка для текущего видеорежима.

Пример – программа BitDemo

Используемая в качестве примера программа `BitDemo` будет просто изображать в окне представления шахматную доску, растягивая или сжимая ее так, чтобы она занимала все окно представления.

Сгенерируем исходный код программы посредством мастера AppWizard, повторяя те же шаги и выбирая те же значения параметров, что и для программы WinGreet (за исключением имени проекта – BitDemo).

В редакторе ресурсов создадим новый ресурс типа Bitmap (идентификатор по умолчанию – IDB_BITMAP1). Создадим рисунок, используя инструменты встроенного редактора.



Альтернативный вариант: создадим это изображение в другом графическом редакторе, сохраним в формате .bmp или .dib, а затем импортируем его, выбрав команду Resource... в меню Insert, а затем нажав кнопку Insert в появившемся окне.

Добавим необходимые переменные в определение класса представления программы (файл BitDemoView.h):

```
class CBitDemoView : public CView
{
protected:
    CBitmap m_Bitmap;
    int m_BitmapHeight;
    int m_BitmapWidth;
// ...
```

Добавим инициализацию этих переменных в конструкторе класса представления (файл BitDemoView.cpp):

```
CBitDemoView::CBitDemoView()
{
    // Здесь добавьте код конструктора
    BITMAP BM;
    m_Bitmap.LoadBitmap (IDB_BITMAP1);
    m_Bitmap.GetObject (sizeof (BM), &BM);
    m_BitmapWidth = BM.bmWidth;
    m_BitmapHeight = BM.bmHeight;
}
```


В конце функции InitInstance (файл BitDemo.cpp) добавим строку:

```
m_pMainWnd->SetWindowText ("Bitmap Demo");
```

Для отображения рисунка добавим код в функцию OnDraw (файл BitDemoView.cpp):

```
void CBitDemoView::OnDraw(CDC* pDC)
{
    CBitDemoDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    // Добавьте код отображения собственных данных
    CDC MemDC;
    RECT ClientRect;
    // Создадим объект памяти КУ и выберем объект изображения
    MemDC.CreateCompatibleDC (NULL);
    MemDC.SelectObject (&m_Bitmap);
    // Получим размеры окна представления
    GetClientRect (&ClientRect);
    // Отобразим рисунок с растяжением или сжатием
    pDC->StretchBlt
        ( 0, 0,                // верхний левый угол приемника
          ClientRect.right,    // ширина прямоугольника приемника
          ClientRect.bottom,   // высота прямоугольника приемника
          &MemDC,              // объект КУ источника
          0, 0,                // верхний левый угол источника
          m_BitmapWidth,       // ширина прямоугольника источника
          m_BitmapHeight,      // высота прямоугольника источника
          SRCCOPY);            // код растровой операции
}
```

Теперь можно скомпилировать приложение и протестировать его.

Тема 17. Печать и предварительный просмотр

17.1. Добавление в программу средств печати и предварительного просмотра

Добавление средств печати при генерации кода

При генерации новой программы основные средства печати и предварительного просмотра можно добавить, если просто установить опцию "Printing and print preview" в диалоговом окне Step4. В этом случае мастер AppWizard добавит соответствующие команды меню и реализацию соответствующих функций. Эта базовая реализация обеспечивает печать части документа, которая помещается на одной странице. Оставшаяся часть документа игнорируется.

Мы здесь рассмотрим процесс добавления средств печати в уже написанную программу. В качестве примера мы возьмем последнюю из разработанных нами версий программы MiniDraw.

Модификация ресурсов программы

Откроем существующий проект MiniDraw и перейдем на вкладку ResourceView. Откроем меню IDR_MAINFRAME в конструкторе меню. Непосредственно под существующей командой Save As... в меню File добавим разделитель и команды Print..., Print Preview и Print Setup. Свойства добавляемых пунктов приведены в следующей таблице.

Идентификатор	Надпись	Интерактивная справка	Другие свойства
—	—	—	Separator
ID_FILE_PRINT	&Print...\tCtrl+P	Print the document	—
ID_FILE_PRINT_PREVIEW	Print Pre&view	Display full pages	—
ID_FILE_PRINT_SETUP	P&rint Setup...	Change the printer and printing options	—

Откроем в редакторе акселераторов таблицу IDR_MAINFRAME и добавим акселератор с идентификатором ID_FILE_PRINT и комбинацией клавиш Ctrl+P.

Теперь в файл определения ресурсов программы необходимо включить дополнительные предопределенные ресурсы. Для этого в меню View выберем команду Resource Includes... и в конце поля Compile-times directives добавим строку

```
#include "afxprint.rc"
```

Тем самым мы предписываем компилятору ресурсов использовать определения, содержащиеся в этом файле (они задают ресурсы, необходимые для новых команд).

Модификация текста программы

1. Добавление обработчиков команд

Обработчик команды Print Setup... не требуется писать самостоятельно, так как класс CWinApp содержит необходимую функцию (OnFilePrintSetup). Однако MFC не добавляет его в схему сообщений, и это необходимо сделать самостоятельно. Откроем файл MiniDraw.cpp и добавим следующий оператор:

```
BEGIN_MESSAGE_MAP(CMiniDrawApp, CWinApp)
//...
ON_COMMAND(ID_FILE_PRINT_SETUP, CWinApp::OnFilePrintSetup)
END_MESSAGE_MAP()
```

Этого достаточно, чтобы при выборе команды Print Setup... вызывалась функция OnFilePrintSetup класса CWinApp. Аналогичным образом в файле реализации класса представления MiniDrawView.cpp добавим обработчики команд Print и Print Preview:

```
BEGIN_MESSAGE_MAP(CMiniDrawView, CScrollView)
//...
ON_COMMAND(ID_FILE_PRINT, CView::OnFilePrint)
ON_COMMAND(ID_FILE_PRINT_PREVIEW, CView::OnFilePrintPreview)
END_MESSAGE_MAP()
```

2. Переопределение функций

Функции-обработчики (OnFilePrint и OnFilePrintPreview) уже реализованы в классе CView. Они в свою очередь вызывают виртуальные функции, реализованные в этом же классе. Их реализация по умолчанию накладывает ряд ограничений на процесс печати. Как будет показано далее, для расширения возможностей печати некоторые из этих функций можно переопределить.

Для реализации же базовых возможностей печати и предварительного просмотра необходимо переопределить только виртуальную функцию OnPreparePrinting.

Для этого вызовем мастера ClassWizard, откроем вкладку Message Maps, выберем CMiniDrawView в списках Class name и Object IDs, а в списке Messages – OnPreparePrinting и добавим функцию. Отредактируем код добавленной функции, удалив вызов функции OnPreparePrinting и добавив свой:

```

BOOL CMiniDrawView::OnPreparePrinting(CPrintInfo* pInfo)
{
    // TODO: Добавьте свой код обработчика
    // УДАЛЕНО: return CScrollView::OnPreparePrinting(pInfo);
    return DoPreparePrinting (pInfo);
}

```

Отметим, что функцию `OnPreparePrinting` было *необходимо создать*, так как ее стандартная версия ничего не выполняет. Ее использование могло бы привести к попытке напечатать или просмотреть документ без наличия корректного объекта контекста устройства.

Функция `DoPreparePrinting` создает контекст устройства, связанный с принтером. Если документ *печатается*, то она отображает диалоговое окно `Print` для управления этим процессом. При *предварительном просмотре* контекст устройства создается, однако диалоговое окно `Print` не отображается.

Объект класса `CPrintInfo` содержит необходимую информацию о печати и переменные для получения или изменения установок принтера. Указатель на него передается во все виртуальные функции выполнения печати.

После вызова функции `OnPreparePrinting` для создания контекста устройства MFC вызывает функцию `OnDraw` класса представления, передавая ей указатель на КУ. Соответственно отображение информации происходит не в окне представления, а на принтере или в окне предварительного просмотра.

На этом все изменения программы `MiniDraw` для поддержки печати закончены.

Добавление средств печати в окно представления класса `CEditView`

Созданная нами ранее программа `MiniEdit` порождала окно представления не от класса `CView`, а от класса `CEditView`. В этом случае MFC и Windows предоставляют большую часть кода для поддержки печати. Перечислим необходимые действия по реализации команд `Print...`, `Print Preview` и `Print Setup`, если при генерации кода не был установлен флажок "Printing and print preview":

- Для реализации команды `Print` ее достаточно добавить в меню `File` с идентификатором `ID_FILE_PRINT` без всякого изменения кода. Стандартная реализация в этом случае позволяет печатать даже документы, занимающие более одной страницы.
- Для реализации команды `Print Preview` ее следует добавить в меню `File` с идентификатором `ID_FILE_PRINT_PREVIEW`, добавить в схему сообщений директиву `include` для подключения файла `Afxprint.rc` и макрос `ON_COMMAND` для класса представления, как это делалось выше.

- Чтобы реализовать команды Print... и Print Preview *не нужно* создавать функцию OnPreparePrinting.
- Процедура реализации команды Print Setup совпадает с описанной только что.

17.2. Усовершенствованная печать

Текущая версия программы MiniDraw печатает (или просматривает) только часть рисунка, помещающуюся на одну страницу.

Здесь мы расширим возможности программы так, чтобы часть рисунка, не поместившаяся на одной странице, печаталась на дополнительных. Это достигается переопределением некоторых виртуальных функций, вызываемых при печати.

Кроме того, мы модифицируем текущую версию функции OnDraw так, чтобы она рисовала ограничивающие линии справа и внизу рисунка только при выводе в окно представления, но не при печати.

Изменение размера рисунка

В старой версии размер рисунка устанавливался в 640 на 480 пикселей. Этого, очевидно, недостаточно, чтобы продемонстрировать многостраничную печать. Заведем константы для ширины и высоты рисунка в файле MiniDrawView.h:

```
//...
const int DRAWWIDTH  = 4000;  // ширина рисунка
const int DRAWHEIGHT = 6000;  // высота рисунка
```

```
class CMiniDrawView : public CScrollView
{
//...
```

В файле MiniDrawView.cpp изменим код функции OnInitialUpdate, чтобы она использовала эти константы вместо явно прописанных значений:

```
void CMiniDrawView::OnInitialUpdate()
{
//...
    SIZE Size = {DRAWWIDTH, DRAWHEIGHT};
    SetScrollSizes (MM_TEXT, Size);
}
```

Чтобы не спутать файлы, созданные разными (несовместимыми) версиями программы, в файле MiniDrawDoc.cpp, изменим номер версии документа. Во всех вхождениях макроса IMPLEMENT_SERIAL (их должно быть 8) вида

```
IMPLEMENT_SERIAL (CFigure, CObject, 2)
```

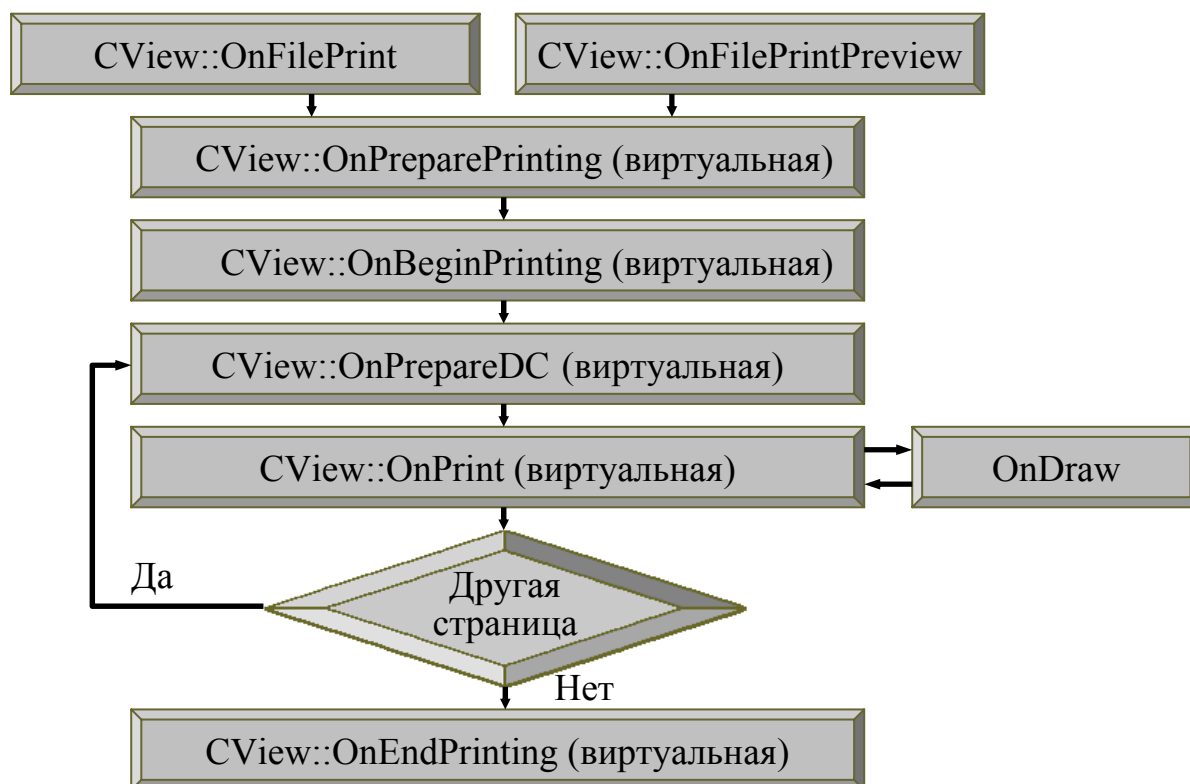
изменим номер версии на следующий:

```
IMPLEMENT_SERIAL (CFigure, CObject, 3)
```

Замечание. Обычно программы рисования позволяют программно изменить размер рисунка и сохраняют его вместе с изображением в файле. Это позволило бы нам не исправлять номер версии при столь незначительной модификации программы.

Схема процесса печати и предварительного просмотра

Следующая блок-схема наглядно представляет порядок вызова функций печати и предварительного просмотра.



Виртуальные функции печати и их назначение

```
virtual BOOL OnPreparePrinting( CPrintInfo* pInfo );
```

Возвращаемое значение. Отлично от нуля, чтобы начинать печатать. 0, если задание по выводу на печать было отменено.

Параметр pInfo – указатель на экземпляр класса CPrintInfo, который описывает текущее задание по выводу на печать.

Описание. Вызывается системой прежде, чем документ напечатан или визуально обследован. Заданная по умолчанию реализация не делает ничего. Необходимо перегрузить эту функцию для предварительного просмотра печати и печати. Вызовите функцию DoPreparePrinting, передавая ей параметр pInfo, и затем возвратите ее результат.

DoPreparePrinting отображает диалоговое окно Print и создает контекст устройства принтера. Если требуется инициализировать диалоговое окно Print со значениями иными, чем значения по умолчанию, задайте их соответствующим элементам **pInfo**. Например, если известна длина документа, это значение следует передать функции **SetMaxPage** класса **CPrintInfo** перед вызовом **DoPreparePrinting**. Оно отображается в диалоговом окне Print.

DoPreparePrinting не отображает диалоговое окно Print в случае вызова для предварительного просмотра. Если Вы не хотите отображать диалоговое окно Print для задания по выводу на печать, проверьте, что член **pInfo** с именем **m_bPreview** равен **FALSE**, затем установите его равным **TRUE** до вызова **DoPreparePrinting**, впоследствии установите его вновь равным **FALSE**.

Если нужно выполнить инициализирующие действия, которые требуют доступа к объекту CDC, представляющему контекст устройства принтера (например, если необходимо знать размер страницы перед определением длины документа), следует перегрузить функцию **OnBeginPrinting**.

Если нужно устанавливать значение переменных-членов класса **CPrintInfo** **m_nNumPreviewPages** или **m_strPageDesc**, это следует делать после вызова **DoPreparePrinting**, так как эта функция устанавливает **m_nNumPreviewPages**, равным значению, найденному в .INI файле прикладной программы, а **m_strPageDesc** устанавливает равным значению по умолчанию.

```
virtual void OnBeginPrinting( CDC* pDC, CPrintInfo* pInfo );
```

Параметры:

pDC – указатель на контекст устройства принтера.

pInfo – указатель на экземпляр класса **CPrintInfo**, который описывает текущее задание по выводу на печать.

Описание. Вызывается системой в начале печати или работы предварительного просмотра печати, после вызова **OnPreparePrinting**. Заданная по умолчанию реализация этой функции не делает ничего. Следует перегрузить эту функцию, если нужно разместить любые GDI-ресурсы, типа перьев или шрифтов, необходимых специально для печати. Выбирать эти ресурсы в контекст устройства следует в функции **OnPrint** для каждой страницы, которая использует их. Если для печати и просмотра используется один и тот же объект представления, разумно использовать разные переменные для GDI -ресурсов, это позволяет изменять изображение на экране во время печати. Вы можете также использовать эту функцию, чтобы выполнить инициализации, которые зависят от реквизитов контекста устройства принтера. Например, число страниц, необходимых чтобы печатать документ, может зависеть от параметров настройки, определенных в диалоговом окне Print (типа длины страницы). В такой ситуации нельзя определять длину документа в функции **OnPreparePrinting**. **OnBeginPrinting**

– первая перегружаемая функция, которая предоставляет доступ к объекту CDC, представляющему контекст устройства принтера, так что те устанавливать длину документа следует из этой функции.

```
virtual void OnPrepareDC( CDC* pDC, CPrintInfo* pInfo=NULL );
```

Параметры:

pDC – указатель на контекст устройства, который нужно использовать для изображения документа.

pInfo – указатель на экземпляр класса CPrintInfo, который описывает текущее задание по выводу на печать. m_nCurPage определяет номер печатаемой страницы. Параметр равен NULL, если отображение направляется на экран.

Описание. Вызывается системой прежде, чем вызывается функция OnDraw для отображения на экране, и прежде, чем для очередной страницы вызывается функция OnPrint. Заданная по умолчанию реализация этой функции не делает ничего, если функция вызвана для отображения на экране. Однако, поскольку эта функция перегружена в производных классах, типа CScrollView, обязателен вызов ее базовой реализации в начале вашей перегруженной функции.

Если длина документа не была определена в pInfo, OnPrepareDC считает, что документ имеет одну страницу и останавливает цикл печати после того, как одна страница была напечатана. Функция останавливает цикл печати, устанавливая значение m_bContinuePrinting равным FALSE.

Перегружать функцию OnPrepareDC требуется в следующих случаях:

- Необходимо корректировать атрибуты контекста устройства для определенной страницы. Например, если требуется установить режим отображения или другие характеристики контекста устройства, делать это следует в функции OnPrepareDC .
- Требуется выполнить разбиение на страницы. Обычно длина документа определяете в начале печати, с использованием функции OnPreparePrinting. Однако, если длина документа заранее неизвестна (например, при печати неопределенного числа записей из базы данных), следует перегрузить функцию OnPrepareDC, чтобы во время печати определить конец документа. Когда выясняется, что больше нет страниц для печати, следует установить значение m_bContinuePrinting (член класса CPrintInfo) равным FALSE.
- Требуется посылать управляющие коды принтеру. Чтобы это сделать из функции OnPrepareDC, следует использовать вызовите функцию Escape – член класса CDC, указателем на который является параметр pDC.


```
virtual void OnPrint( CDC* pDC, CPrintInfo* pInfo );
```

Параметры:

pDC – указатель на контекст устройства принтера.

pInfo – указатель на экземпляр класса CPrintInfo, который описывает текущее задание по выводу на печать.

Описание. Вызывается системой, чтобы напечатать или предварительно показать страницу документа. Для каждой страницы эта функция вызывается сразу после вызова функции OnPrepareDC. Номер страницы определяется значением m_nCurPage – членом класса CPrintInfo (указатель pInfo). Заданная по умолчанию реализация вызывает функцию OnDraw и передает ей контекст устройства принтера.

Перегружать функцию OnPrint требуется в следующих случаях:

- Требуется выполнить печать многостраничного документа. Функция должна выполнить печать только части документа, соответствующей текущей странице. Если для печати используется вызов функции OnDraw, можно скорректировать начало области просмотра так, чтобы была напечатана только соответствующая часть документа.
- Требуется, чтобы печатный вариант изображения был отличен от изображения на экране (то есть программа – не WYSIWYG). Вместо того, чтобы передавать контекст устройства принтера функции OnDraw, следует выполнить отображение с использованием атрибутов контекста устройства, не используемых при отображении на экране. Если для печати требуются специальные GDI-ресурсы, их следует выбрать в контекст устройства перед отображением и восстановить измененные значения после этого. Эти GDI-ресурсы должны быть размещены в OnBeginPrinting и освобождены в OnEndPrinting.
- Печатать верхние колонтитулы или нижние колонтитулы. В этом случае можно использовать OnDraw для вывода на принтер, ограничивая соответствующим образом область печати.

Обратите внимание, что член класса CPrintInfo (указатель pInfo) m_rectDraw задает размеры печатаемой области страницы в логических единицах.

Не следует вызывать OnPrepareDC в перегруженной функции OnPrint, так как система вызывает OnPrepareDC автоматически перед вызовом OnPrint.

```
virtual void OnEndPrinting ( CDC* pDC, CPrintInfo* pInfo );
```

Параметры:

pDC – указатель на контекст устройства принтера.

pInfo – указатель на экземпляр класса CPrintInfo, который описывает текущее задание по выводу на печать.

Описание. Вызывается системой после того, как документ был напечатан или предварительно показан. Заданная по умолчанию реализация этой функции не делает ничего. Перегружать эту функцию требуется, если нужно освободить любые GDI-ресурсы, которые были размещены в функции OnBeginPrinting. Для этой цели следует использовать функцию CGdiObject::DeleteObject.

Переопределение виртуальных функций печати

Ранее нами была переопределена функция OnPreparePrinting. Для реализации многостраничной печати нам потребуется также переопределить функции OnBeginPrinting и OnPrepareDC.

Сначала требуется добавить в определение класса представления CMiniDrawView несколько новых переменных (файл MiniDrawView.h):

```
class CMiniDrawView : public CScrollView
{
// ...
    HCURSOR m_HCross;
    int m_NumCols, m_NumRows;
    int m_PageHeight, m_PageWidth;
    CPen m_PenDotted;
// ...
}
```

Переменные m_NumCols и m_NumRows используются для хранения количества страниц по горизонтали и вертикали, а переменные m_PageHeight и m_PageWidth – для хранения размеров страниц.

Переопределение виртуальной функции OnBeginPrinting

Для переопределения виртуальной функции OnBeginPrinting вызовем мастера ClassWizard, откроем вкладку Message Maps, выберем CMiniDrawView в списках Class name и Object IDs, а в списке Messages – имя функции OnBeginPrinting и нажмем кнопку Add Function. Аналогичным образом добавим переопределение функции OnPrepareDC.отредактируем код добавленной функции OnBeginPrinting следующим образом (файл MiniDrawView.cpp):

```
void CMiniDrawView::OnBeginPrinting(CDC* pDC,
                                     CPrintInfo* pInfo)
{
    // TODO: Добавьте свой код обработчика
    // Получить размеры доступной для печати области страницы
    m_PageHeight = pDC->GetDeviceCaps (VERTRES); // в пикселях
    m_PageWidth = pDC->GetDeviceCaps (HORZRES); // в пикселях
    m_NumRows = DRAWHEIGHT / m_PageHeight +
                (DRAWHEIGHT % m_PageHeight > 0);
}
```

```

m_NumCols = DRAWWIDTH / m_PageWidth +
            (DRAWWIDTH % m_PageWidth > 0);
pInfo->SetMinPage (1);
pInfo->SetMaxPage (m_NumRows * m_NumCols);
CScrollView::OnBeginPrinting(pDC, pInfo);
}

```

В приведенном коде вызов функции `CPrintInfo::SetMinPage` устанавливает номер первой печатаемой страницы, а вызов `CPrintInfo::SetMaxPage` — общее количество выводимых страниц. MFC вызовет функции `OnPrepareDC` и `OnPrint` необходимое количество раз. В диалоговом окне `Print` можно задать номера страниц внутри установленного таким образом диапазона. Если не вызывалась функция `SetMaxPage`, то в функции `OnPrepareDC` следует предусмотреть ручное прерывание цикла печати путем установки значения `CPrintInfo::m_bContinuePrinting` в `FALSE`.

Переопределение виртуальной функции `OnPrepareDC`

Отредактируем теперь код функции `OnPrepareDC` (файл `MiniDrawView.cpp`):

```

void CMiniDrawView::OnPrepareDC(CDC* pDC, CPrintInfo* pInfo)
{
    // TODO: Добавьте собственный код обработчика
    CScrollView::OnPrepareDC(pDC, pInfo);
    if (pInfo == NULL)
        return;
    int CurRow = pInfo->m_nCurPage / m_NumCols +
                (pInfo->m_nCurPage % m_NumCols > 0);
    int CurCol = (pInfo->m_nCurPage - 1) % m_NumCols + 1;
    pDC->SetViewportOrg (-m_PageWidth * (CurCol-1),
                        -m_PageHeight * (CurRow-1));
}

```

Как отмечалось ранее, MFC вызывает функцию `OnPrepareDC` перед печатью каждой страницы. Однако MFC вызывает эту функцию *также* и непосредственно перед вызовом `OnDraw` для перерисовки окна представления. Поскольку в нашем случае окно представления поддерживает прокрутку (порождается от класса `CScrollView`), возникает необходимость согласовать начальную позицию просмотра с текущей позицией прокрутки документа.

Если функция `OnPrepareDC` базового класса вызывается для перерисовки окна представления, то этот вызов настраивает объект контекста устройства на текущую позицию прокрутки. После этого указателю `pInfo` присваивается значение `NULL`. В этом случае наш вариант функции ничего больше не делает.

Если `pInfo` не равно `NULL`, то это адрес объекта `CPrintInfo`, и наш код настраивает объект контекста устройства так, чтобы функция `OnDraw`

печатала *следующую* часть рисунка на текущей странице (аналогично методу прокрутки документа в окне представления, рассмотренному ранее).

Переменная `m_nCurPage` класса `CPrintInfo` содержит номер текущей печатаемой страницы. Это значение вместе с переменной `m_NumCols`, установленной функцией `OnBeginPrinting`, используется для вычисления строки (`CurRow`) и столбца (`CurCol`) позиции части рисунка, выводимой на печать. Далее эти значения используются для вычисления новых координат начала представления, передаваемых в функцию `SetViewportOrg` класса `CDC`.

Модификация виртуальной функции `OnDraw`

После вызова функции `OnPrepareDC` MFC вызывает функцию `OnPrint`. Стандартная реализация этой функции просто вызывает функцию `OnDraw` класса представления программы, передавая ей указатель на объект контекста устройства, созданный функцией `OnPreparePrinting` и подготовленный функцией `OnPrepareDC`. Поскольку начало представления согласовано функцией `OnPrepareDC` с объектом контекста устройства, функция `OnDraw` автоматически печатает корректную часть рисунка.

В нашем переопределении мы только предотвратим вывод линий границы при печати, поместив их в блок `if`:

```
void CMiniDrawView::OnDraw(CDC* pDC)
{
    CMiniDrawDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    // TODO: добавьте код отображения собственных данных
    if (pDC->GetDeviceCaps(TECHNOLOGY) == DT_RASDISPLAY)
    {
        CSize ScrollSize = GetTotalSize();
        pDC->MoveTo(ScrollSize.cx, 0);
        pDC->LineTo(ScrollSize.cx, ScrollSize.cy);
        pDC->LineTo(0, ScrollSize.cy);
    }
    // ...
}
```

Функция `GetDeviceCaps`

Если функции `GetDeviceCaps`, как в этом примере, передать значение константы `TECHNOLOGY`, то она возвратит код, указывающий тип устройства, связанного с объектом контекста устройства. В нашем случае это значение проверяется на равенство `DT_RASDISPLAY`, что означает, что это устройство – экран.

Обычно функция `GetDeviceCaps` вызывается из переопределенных функций `OnBeginPrinting`, `OnPrepareDC` или `OnPrint`, либо из стандартной функции `OnDraw` класса представления. Мы в примерах уже использовали

ее для определения типа устройства вывода (аргумент TECHNOLOGY), размер области печати в пикселях (аргументы HORZSIZE и VERTSIZE).

При выводе на экран предполагается, что дисплей поддерживает все основные операции рисования и отображения. Однако принтер или плоттер не обязательно их поддерживают. Для выяснения, поддерживает ли устройство некоторые растровые или битовые операции, также можно использовать эту функцию. Пример:

```
int Caps = pDC->GetDeviceCaps(RASTERCAPS);
    if (Caps & RC_BITBLT)
        // Теперь можно вызывать PatBlt и BitBlt
    if (Caps & RC_STRETCHBLT)
        // Теперь можно вызывать StretchBlt
```

Также эту функцию можно вызывать для определения возможности принтера при рисовании некоторых изображений, например, рисование кривых (аргумент CURVECAPS) или многоугольников (аргумент POLYGONCAPS). Более подробную информацию можно получить из документации.

Тема 18. Многопоточные приложения

Выполняемую программу, обладающую собственной памятью, описателями файлов и другими системными ресурсами, принято называть *процессом* (process). Процесс может допускать несколько параллельных путей исполнения кода, называемых *потоками* (threads).

При этом функции не привязаны к потокам – одну и ту же функцию могут вызывать несколько потоков. По большей части, все пространство кода и данных процесса доступно всем его потокам. Потоками управляет операционная система, и у каждого потока есть свой стек.

Потоки в Windows бывают двух видов: *рабочие потоки* (worker threads) и *потоки пользовательского интерфейса* (user-interface threads). MFC поддерживает оба вида потоков. У потока пользовательского интерфейса есть окна, а значит, и свой цикл обработки сообщений, а у рабочего – нет (хотя можно и организовать). Рабочие потоки легче программировать, и они, как правило, полезнее. Ниже будет рассмотрено программирование именно рабочих потоков.

Даже в однопоточном приложении есть поток, который называется *основным потоком* (main thread). Собственно приложение – это и есть поток.

В данной теме мы рассмотрим вопрос о создании вторичных потоков и управлении ими. Будут также рассмотрены механизмы синхронизации работы потоков, предоставляемые Win32 API. В качестве примера будет создана многопоточная версия программы Mandel.

18.1. Создание и управление вторичными потоками

В программах с графическим интерфейсом разумно организовывать работу потоков таким образом, чтобы основной (первичный) поток занимался обработкой сообщений, что позволяло бы программе быстро реагировать на поступающие команды и другие события.

При этом вторичный поток (или потоки) используется для выполнения длительных задач, которые могли бы блокировать на время обработку сообщений, например, при рисовании сложных графических изображений, пересчете электронных таблиц, выполнении дисковых операций, работе с последовательным портом и т.п.

Запуск отдельного потока происходит быстрее, чем запуск процесса, и требует незначительного объема памяти. Кроме того, как было отмечено, все потоки имеют доступ к основной части ресурсов программы.

Настройка среды разработчика.

Прежде всего, следует убедиться, что установка проекта Use run-time library имеет значение, соответствующее многопоточному режиму. Для этого в меню Project выберем команду Settings... и имя проекта в иерархи-

ческом списке проектов (в левой части окна). Откроем вкладку C/C++ и выберем пункт Code Generation в списке Category. В списке Use run-time library выберем необходимое значение: Multithreaded или Multithreaded DLL (для отладочной версии: Debug Multithreaded или Debug Multithreaded DLL).

Создание и запуск вторичного потока

Для запуска нового потока используется глобальная MFC-функция

```
CWinThread* AfxBeginThread
```

```
( AFX_THREADPROC pfnThreadProc,  
  LPVOID pParam,  
  int nPriority = THREAD_PRIORITY_NORMAL,  
  UINT nStackSize = 0,  
  DWORD dwCreateFlags = 0,  
  LPSECURITY_ATTRIBUTES lpSecurityAttrs = NULL );
```

Функция `AfxBeginThread` инициализирует библиотеку MFC для работы в многопоточном режиме, потом вызывает функцию `_beginthreadex` библиотеки периода выполнения, а затем для запуска потока вызывает функцию Win32 API `::CreateThread`. Это позволяет без проблем использовать классы MFC.

В MFC-программах *нельзя* создавать поток, непосредственно вызывая функции `_beginthread`, `_beginthreadex` или `::CreateThread`.

Функция `AfxBeginThread` запускает новый поток и сразу возвращает управление. С этого момента оба потока существуют одновременно. Первый параметр (`pfnThreadProc`) задает функцию, которой передается управление в новом потоке, а второй (`pParam`) – значение, передаваемое этой функции.

Функция потока должна иметь следующий формат:

```
UINT ThreadFunction (LPVOID pParam);
```

Возвращаемое ею значение является *кодом возврата*. Обычно при нормальном завершении функция возвращает нулевое значение. Не разрешается возвращать специальное значение `STILL_ACTIVE (0x00000103L)`, означающее, что поток все еще выполняется.

Обычно `pParam` – это указатель на структуру, которая содержит всю передаваемую информацию.

Четыре последних параметра функции `AfxBeginThread` имеют стандартные значения, поэтому их можно опустить при запуске потока.

Параметр `nPriority` задает *приоритет* потока, который определяет, как часто поток будет получать управление. Обычно ему присваивается значение, соответствующее среднему приоритету – `THREAD_PRIORITY_NORMAL`.

Параметр `nStackSize` определяет размер стека. Значение 0 (используется в большинстве случаев) означает, что размер стека устанавливается по умолчанию.

Если параметр `dwCreateFlags` равен 0 (стандартное значение), то новый поток сразу становится активным. Если ему присвоить значение `CREATE_SUSPEND`, то поток нужно будет активизировать явно, вызвав функцию `CWinThread::ResumeThread`.

Параметр `lpSecurityAttrs` определяет атрибуты защиты. Значение 0 (используется в большинстве случаев) означает, что атрибуты защиты устанавливаются по умолчанию.

Прекращение выполнения потока

Наиболее естественный способ завершения потока – это возврат из его главной функции через оператор `return` с указанием кода завершения. При этом стек потока освобождается, все автоматические данные корректно уничтожаются, т.е. для всех экземпляров классов вызываются деструкторы.

Альтернативный способ завершения – вызов потоком MFC-функции `AfxEndThread` с передачей ей кода завершения в качестве параметра:

```
void AfxEndThread (UINT nExitCode);
```

При этом стек потока освобождается, однако деструкторы для автоматических объектов *не вызываются*.

Замечание. Оба способа предполагают, что соответствующие действия выполняются *самим потоком*. Если есть необходимость завершить поток из *другого* потока, то самое правильное решение – передать сигнал в завершаемый поток, по которому он себя завершает сам. Допустимо также использование способа, рекомендуемого документацией, а именно использование функции `::TerminateThread`, предоставляемой Win32 API.

Управление потоком

Функция `AfxBeginThread` возвращает указатель на объект класса `CWinThread`, который можно использовать для управления потоком. Его следует сохранить, например:

```
int Value = 1;  
CWinThread *pWinThread;  
pWinThread = AfxBeginThread (ThreadFunction, &Value);
```

Для временной приостановки выполнения потока родительский поток может вызвать функцию `CWinThread::SuspendThread`:

```
pWinThread -> SuspendThread ();
```

Для возобновления выполнения потока используется функция `ResumeThread`:

```
pWinThread -> ResumeThread ();
```


Если функция `SuspendThread` вызывалась несколько раз, то столько же раз для возобновления работы потока требуется вызвать функцию `ResumeThread`.

Можно изменить приоритет потока, вызвав функцию `SetThreadPriority`, например:

```
pWinThread ->
    SetThreadPriority(THREAD_PRIORITY_ABOVE_NORMAL);
```

Текущее значение приоритета возвращает функция `GetThreadPriority`.

Родительский поток может использовать функцию `::GetExitCodeThread`, чтобы определить, продолжается ли выполнение потока:

```
DWORD ExitCode;
::GetExitCodeThread (pWinThread->m_hThread, &ExitCode);
if (ExitCode == STILL_ACTIVE)
    // поток продолжает выполнение
else
    // поток завершен, код возврата в переменной ExitCode
```

Использование данного кода может вызвать проблему, заключающуюся в том, что если поток создан с установками по умолчанию, при его завершении объект класса `CWinThread` *автоматически удаляется*, и обращение к нему вызовет ошибку. Чтобы предотвратить ее, можно использовать такой код создания потока:

```
pWinThread = AfxBeginThread (ThreadFunction, &Value,
    THREAD_PRIORITY_NORMAL, 0, CREATE_SUSPEND);
pWinThread -> m_bAutoDelete = FALSE;
pWinThread -> ResumeThread ();
```

В этом случае после окончания работы с объектом потока его следует удалить явно:

```
delete pWinThread;
```

18.2. Особенности использования MFC-классов в многопоточных программах

Ограничения на использование MFC-классов

Потоки, созданные в одном процессе, выполняются в одном адресном пространстве, совместно используют глобальные и динамические переменные и объекты. Однако существует ряд ограничений на использование объектов MFC-классов.

Во-первых, два потока не могут одновременно получать доступ к одному объекту MFC. Например, они могут использовать один и тот же объект класса `CString`, но не могут одновременно вызывать его функции. Чтобы предотвратить такое некорректное использование объектов MFC, сле-

дует воспользоваться одним из способов синхронизации, рассматриваемых далее.

Во-вторых, объекты некоторых классов, а также объекты классов, производных от них, доступны только для потока, их создавшего. Это классы:

- CWnd,
- CDC,
- CMenu,
- CGdiObject.

Причина в том, что каждый из таких объектов хранит дескриптор для некоторых подчиненных классов. Дочерний поток должен, получив этот дескриптор, сам создать соответствующий объект для работы с ним. Например, поток создает объект класса CWnd для управления окном. Вместо передачи этого объекта в порождаемый поток ему следует передать дескриптор объекта, хранящийся в переменной m_hWnd объекта. Тот, в свою очередь, создает собственный объект класса CWnd, вызывая функцию FromHandle этого класса. Пример будет приведен ниже.

Создание потоков пользовательского интерфейса

Как мы отмечали ранее, чаще используются *рабочие потоки*, позволяющие производить вычисления, пока главный поток занят обработкой сообщений.

Для создания *потока пользовательского интерфейса* следует вызвать вторую версию функции AfxBeginThread, использующей указатель CRuntimeClass в качестве первого параметра:

```
CWinThread* AfxBeginThread
( CRuntimeClass* pThreadClass,
  int nPriority = THREAD_PRIORITY_NORMAL,
  UINT nStackSize = 0,
  DWORD dwCreateFlags = 0,
  LPSECURITY_ATTRIBUTES lpSecurityAttrs = NULL );
```

Такой поток может создавать окна и обрабатывать сообщения, передаваемые в них. Сначала требуется построить класс для управления потоком, произведенный от CWinThread, и переопределить в нем виртуальную функцию InitInstance класса CWinThread. Она должна создавать все окна, отображаемые потоком, и выполнять все инициализации. Она очень похожа на одноименную функцию класса приложения, который фактически управляет первичным потоком программы.

Более детальную информацию можно найти в соответствующих разделах справочной системы.

18.3. Синхронизация потоков

Для 32-разрядных программ, выполняющихся в среде Windows 95+ или различных версий Windows NT, периодически происходит *переключение контекста* (context switching) по некоторому алгоритму, причем различному для разных версий Windows. Выполнение любого потока может быть прервано в произвольный момент времени, и управление перейдет к другому потоку. Потоки выполняются *асинхронно*, так что невозможно сказать, какой оператор другого потока выполняется в данный момент.

В таких условиях псевдопараллельная работа потоков с общими данными может привести к недетерминированному поведению программы, что, как правило, нежелательно.

Для осуществления синхронизации действий потоков при работе с общими данными Win32 API содержит набор специальных *объектов синхронизации*. С их помощью можно организовать:

- запрет одновременного доступа нескольких потоков к общим ресурсам;
- ограничение количества потоков, одновременно работающих с ресурсом;
- пересылку сигналов между потоками.

MFC также предоставляет набор классов для объектов синхронизации. Однако они менее понятны, чем объекты Win32 API, и, кроме того, имеют по сравнению с последними некоторые ограничения по использованию. Поэтому в дальнейшем мы ограничимся рассмотрением средств синхронизации Win32 API.

Использование мьютексов

Мьютекс – один из наиболее простых и типичных объектов синхронизации. Название его происходит от выражения *mutual exclusion* (*взаимное исключение*). Используется он для исключения использования ресурса более чем одним потоком. Для создания этого объекта используется функция `::CreateMutex`. Например:

```
HANDLE hMutex; // Глобальная переменная, доступна всем потокам
// ...
void SomeFunction ()
{
    // ...
    hMutex = ::CreateMutex
        ( NULL,    // стандартные атрибуты защиты
          FALSE,   // мьютекс изначально свободен
          NULL ); // имя мьютексу не присваивается
    // ...
}
```

Функция `CreateMutex` возвращает дескриптор, используемый в дальнейшем всеми потоками для ссылки на мьютекс.

Далее при доступе к общему ресурсу во всех потоках следует добавить вызов функции `::WaitForSingleObject`. Например:

```
::WaitForSingleObject
    ( HMutex,      // дескриптор мьютекса
      INFINITE); // ждать столько, сколько нужно
// Работа с общим ресурсом
::ReleaseMutex (HMutex);
```

Второй параметр функции `CreateMutex` задает время ожидания в миллисекундах. По истечении этого времени функция возвращает управление, даже если мьютекс не перешел в свободное состояние. Понять причину возврата управления из функции `WaitForSingleObject` можно, если сохранить и проанализировать возвращаемое ею значение типа `DWORD`. В случае выхода по завершении тайм-аута это значение будет равно `WAIT_TIMEOUT`.

Отметим также, что использование значения `INFINITE` в качестве параметра не всегда безопасно: если объект так и не перейдет в свободное состояние, поток никогда не проснется.

Функцию `::WaitForMultipleObjects` можно использовать для ожидания освобождения нескольких мьютексов – всех сразу или одного любого.

По завершении использования мьютекса его следует закрыть:

```
::CloseHandle (HMutex);
```

Другие объекты синхронизации

Кроме мьютексов, в системе Win32 можно использовать и некоторые другие объекты синхронизации: критические секции, семафоры и события.

Критические секции. Выполняют те же задачи, что и мьютексы. Для них вызываются другие функции Win32. Они немного эффективнее, но не позволяют синхронизировать работу потоков в различных процессах.

Семафоры. В отличие от мьютексов допускают одновременный доступ к ресурсам *нескольких* потоков. При создании семафора указывается максимальное число таких потоков.

События. Событие – многофункциональный объект синхронизации, позволяющий потокам обмениваться сигналами.

Объект синхронизации	Назначение объекта	Функции Win32 API
Мьютекс	Запрет доступа нескольким потокам к общим ресурсам	::CreateMutex ::WaitForSingleObject ::WaitForMultipleObjects ::ReleaseMutex ::CloseHandle
Критическая секция	Запрет доступа нескольким потокам к общим ресурсам. Эффективно, но не может быть использовано разными процессами.	::InitializeCriticalSection ::EnterCriticalSection ::LeaveCriticalSection ::DeleteCriticalSection
Семафор	Ограничение числа потоков, которые могут иметь одновременный доступ к общим ресурсам.	::CreateSemaphore ::WaitForSingleObject ::WaitForMultipleObjects ::ReleaseSemaphore ::CloseHandle
Событие	Передача потоком сигналов одному или нескольким другим потокам	::CreateEvent ::SetEvent ::PulseEvent ::ResetEvent ::WaitForSingleObject ::WaitForMultipleObjects ::CloseHandle

Другие типы синхронизации

Функции ::WaitForSingleObject и ::WaitForMultipleObjects могут использоваться для ожидания доступа к объектам Windows и других типов. В этом контексте *объектом* мы называем не экземпляр класса, а элемент Windows, представленный дескриптором. В таблице представлены некоторые объекты, допускающие синхронизацию подобного рода.

Объект Windows	Функция или переменная для получения дескриптора объекта	Событие, по которому объект становится свободным
Поток	CWinThread::m_hThread	Прекращение потока
Процесс	::CreateProcess, ::OpenProcess	Прекращение процесса
Ввод с клавиатуры	::CreateFile, ::GetStdHandle	Доступен ввод с клавиатуры
Обмен уведомлениями	::FindFirstChangeNotification	Обмен в специальном каталоге

Кроме того, поток может вызвать функцию `::MsgWaitForMultipleObjects` для ожидания либо освобождения одного или более объектов, либо получения одного или более указанных сообщений. Можно воспользоваться функцией `::Sleep` для ожидания в течение заданного количества миллисекунд. Обе эти функции блокируют поток, так что он не занимает процессорного времени. Наконец, для синхронизации можно воспользоваться глобальными переменными.

Многопоточная программа MandelMT

В разработанной нами ранее программе Mandel первичный (и единственный) поток программы в цикле рисовал один столбец изображения, затем обрабатывал поступившие сообщения и так далее.

Наша цель – разработать версию этой программы, позволяющую программе отвечать на сообщения, *пока* выводится изображение. Первичный поток программы будет отвечать за обработку сообщений, а вторичный – формировать изображение.

Этот подход лучше для программирования еще и потому, что каждый поток выполняет только собственную задачу, не переключаясь на другие.

Вместе с тем этот подход порождает и некоторые проблемы. Так, необходимо запретить вторичному потоку рисование в окне, пока первичный передвигает или модифицирует его. Кроме того, как мы отмечали ранее, имеется ряд ограничений по использованию MFC-классов.

Для создания многопоточной версии программы Mandel мы вместо модификации старой программы сгенерируем новый набор файлов. Для этого, как обычно, воспользуемся мастером AppWizard, проделав все те же действия, что и для программы WinGreet. Имя проекта – MandelMT.

Далее описываются необходимые модификации в сгенерированных файлах исходного кода.

В файле MandelMTView.h определим переменную `m_PDrawThread` класса представления `CMandelMTView`:

```
class CMandelMTView : public CView
{
public:
    CWinThread *m_PDrawThread;
    // ...
```

Эта переменная будет использоваться для хранения адреса объекта, управляющего вторичным потоком. В файле MandelMTView.cpp добавим инициализацию `m_PDrawThread` в конструкторе класса `CMandelMTView`, а в деструкторе – удаление объекта потока:

```
CMandelMTView::CMandelMTView()
{
    // TODO: добавьте код конструктора
    m_PDrawThread = 0;
}
```

```
CMandelMTView::~CMandelMTView()
{
    delete m_PDrawThread;
}
```

В начало файла MandelMTView.cpp добавим следующие определения:

```
// ...
#endif

// Набор констант для построения изображения:
#define CMAX 1.2
#define CIMIN -1.2
#define CRMAX 1.0
#define CRMIN -2.0
#define NMAX 128

// Глобальные переменные для связи между потоками:
int ColMax;
int RowMax;
BOOL StopDraw;
// ...
```

Здесь набор констант для построения изображения остался тем же, что и в предыдущей версии программы. Для хранения текущих размеров окна используются переменные ColMax и RowMax. Они сделаны глобальными, чтобы оба потока имели к ним доступ. Переменная StopDraw – это флажок, установка которого означает для вторичного потока приказ немедленного возврата без завершения рисования.

Создадим обработчик сообщения WM_SIZE. В окне мастера Class Wizard откроем вкладку Message Maps и выберем класс CMandelMTView в списках Class name и Object IDs. Выберем WM_SIZE в списке Messages и добавим обработчик (имя по умолчанию – OnSize). Отредактируем ее код следующим образом:

```
void CMandelMTView::OnSize(UINT nType, int cx, int cy)
{
    CView::OnSize(nType, cx, cy);
    // TODO: Добавьте собственный код обработчика
    if (cx <= 1 || cy <= 1)
        return;
    ColMax = cx;
    RowMax = cy;
}
```

Функция OnSize получает управление при первом создании окна и при изменении его размеров. Мы запоминаем их в переменных ColMax и RowMax.

Теперь добавим в функцию OnDraw файла MandelMTView.cpp необходимый исходный код. Теперь эта функция не занимается рисованием – она только запускает вторичный поток и тут же возвращает управление. Таким образом, программа может продолжить обработку сообщений одновременно с рисованием.

```
void CMandelMTView::OnDraw(CDC* pDC)
{
    CMandelMTDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);

    // TODO: Добавьте код отображения собственных данных
    if (m_PDrawThread)
    {
        StopDraw = TRUE;
        m_PDrawThread->ResumeThread();
        ::WaitForSingleObject
            (m_PDrawThread->m_hThread,    // ждать завершения
             INFINITE);                  // столько, сколько надо
        delete m_PDrawThread;
    }

    m_PDrawThread = AfxBeginThread
        (DrawFractal,
         &m_hWnd,
         THREAD_PRIORITY_BELOW_NORMAL,
         0,
         CREATE_SUSPENDED);
    m_PDrawThread->m_bAutoDelete = FALSE;
    StopDraw = FALSE;
    m_PDrawThread->ResumeThread();
}
```

Функция OnDraw сначала проверяет, был ли уже запущен поток. Если это так, то она устанавливает значение флажка StopDraw в TRUE, вызывает функцию ResumeThread на случай, если поток был приостановлен, дожидается его завершения и удаляет объект. После этого (а также, если поток не был запущен) она создает и запускает новый поток, сохраняя его дескриптор в переменной m_PDrawThread.

Добавим в файл MandelMTView.cpp код глобальной функции DrawFractal:

```
UINT DrawFractal (LPVOID PHWndView)
{
    CClientDC ClientDC (CWnd::FromHandle (*(HWND *)PHWndView));
    int Col, Row;
```



```

static DWORD ColorTable [6] =
    {0x0000ff, // красный
     0x00ff00, // зеленый
     0xff0000, // синий
     0x00ffff, // желтый
     0xffff00, // бирюзовый
     0xff00ff}; // сиреневый
int ColorVal;
float CI, CR = (float)CRMIN;
float DCI = (float)((CIMAX - CIMIN) / (RowMax-1));
float DCR = (float)((CRMAX - CRMIN) / (ColMax-1));
float I, ISqr;
float R, RSqr;
for (Col = 0; Col < ColMax; ++Col)
{
    if (StopDraw)
        break;
    CI = (float)CIMAX;
    for (Row = 0; Row < RowMax; ++Row)
    {
        R = I = RSqr = ISqr = (float)0.0;
        ColorVal = 0;
        while (ColorVal < NMAX && RSqr + ISqr < 4)
        {
            ++ColorVal;
            RSqr = R * R;
            ISqr = I * I;
            I *= R;
            I += I + CI;
            R = RSqr - ISqr + CR;
        }
        ClientDC.SetPixelV(Col, Row, ColorTable[ColorVal%6]);
        CI -= DCI;
    }
    CR += DCR;
}
return (0);
}

```

Вторичный поток выполняет функцию DrawFractal. Она возвращает управление, когда нарисовано все изображение, либо значение флажка StopDraw установлено в TRUE. При возвращении ею управления вторичный поток прерывается.

Так как функция не является членом класса представления, для передачи конструктору класса CClientDC она должна получать указатель на временный объект окна представления через вызов функции FromHandle:

```
CClientDC ClientDC (CWnd::FromHandle (*(HWND *)PHwndView));
```

Именно по этой причине функция OnDraw передает функции DrawFractal не указатель на объект окна представления, а дескриптор Windows.

Теперь мы должны решить еще одну проблему, связанную с рисованием. При перемещении или изменении размеров окна вторичный поток продолжает рисование, а первичный выполняет упомянутые операции. В результате после перемещения или изменения размеров окна изображение будет содержать пробелы. Проблему легко решить путем приостановки вторичного потока на время перемещения или изменения размеров окна. Для этого нам потребуется создать обработчик сообщения WM_SYSCOMMAND.

Замечание. Если установить в настройках рабочего стола Windows опцию отображения содержимого окна при перемещении, указанного неприятного эффекта не будет.

Для добавления обработчика сообщения WM_SYSCOMMAND в окне мастера ClassWizard откроем вкладку Class Info, в списке Class name выберем класс CMainFrame, а в списке Message filter – элемент Window. Это позволит добавлять обработчики для большого числа сообщений, включая и WM_SYSCOMMAND.

Теперь откроем вкладку Message Maps, в списках Object IDs и Class name выберем класс CMainFrame, а в списке Message – пункт WM_SYSCOMMAND. Добавим функцию-обработчик (имя по умолчанию – OnSysCommand). Отредактируем ее код (в файле MainFrm.cpp) следующим образом:

```
void CMainFrame::OnSysCommand(UINT nID, LPARAM lParam)
{
    // TODO: Добавьте собственный код обработчика
    CWinThread *PDrawThread =
        ((CMandelMTView *)GetActiveView ())->m_PDrawThread;

    if (PDrawThread)
        PDrawThread->SuspendThread ();

    CFrameWnd::OnSysCommand(nID, lParam);

    if (PDrawThread && PDrawThread->m_hThread != NULL)
        PDrawThread->ResumeThread ();
}
```

Кроме этого, в файле MainFrm.cpp нужно добавить директивы #include для двух файлов заголовков, чтобы функция OnSysCommand могла получить доступ к определению класса представления:

```
#include "stdafx.h"
#include "MandelMT.h"

#include "MainFrm.h"
#include "MandelMTDoc.h"
#include "MandelMTView.h"
```

```
// ...
```

Завершая работу над программой MandelMT, удалим ненужные команды из меню программы, создадим для нее индивидуальный значок и добавим обычный вызов в функцию InitInstance (файл MandelMT.cpp) для вывода заголовка окна:

```
// ...
m_pMainWnd->SetWindowText("Multithreading Mandelbrot Demo");
return TRUE;
}
```

Протестируем работу программы.

Тема 19. Связи между процессами

19.1. Запуск новых процессов

Функция `::CreateProcess`

Процесс можно определить как исполняемый экземпляр программы. Запуск отдельного процесса медленнее и расходует больше системных ресурсов, чем запуск потока, а обмен информацией между процессами сложнее, чем обмен между потоками. Каждый процесс выполняется в собственном адресном пространстве, что делает маловероятным случайное негативное воздействие одного процесса на работу другого. Для запуска нового процесса используется глобальная MFC-функция `::CreateProcess`.

```
BOOL CreateProcess(  
    LPCTSTR lpszAppName,           // путь к файлу  
    LPTSTR lpszCommandLine,       // командная строка  
    LPSECURITY_ATTRIBUTES lpsaProc, // атр.защиты процесса  
    LPSECURITY_ATTRIBUTES lpsaThread, // атр.защиты потока  
    BOOL bInheritHandles,         // наслед. дескрипторов  
    DWORD dwCreationFlags,        // флаги создания  
    LPVOID lpvEnvironment,       // блок окружения  
    LPCTSTR lpszCurDir,          // текущая директория  
    LPSTARTUPINFO lpsiStartupInfo, // компоненты окна  
    LPPROCESS_INFORMATION lppiProcInfo // информ. о процессе  
);
```

Функция `::CreateProcess` используется для выполнения программ любого типа, поддерживаемого операционной системой, в том числе 16-разрядных программ для Windows 3.1 или MS DOS. Пример использования:

```
STARTUPINFO StartupInfo;  
memset (&StartupInfo, 0,           // использует стандартные  
        sizeof(STARTUPINFO));      // компоненты окна  
StartupInfo.cb=sizeof(STARTUPINFO); // требуется заполнить  
PROCESS_INFORMATION ProcessInfo;  
::CreateProcess(  
    "DoIt.exe", // запустить исполняемый файл "DoIt.exe"  
    NULL,       // командная строка не задана  
    NULL,       // стандартные атрибуты защиты процесса  
    NULL,       // стандартные атрибуты защиты потока  
    FALSE,      // процесс не наследует дескрипторы  
    0,          // стандартные флаги создания  
    NULL,       // использует среду родительского процесса  
    NULL,       // текущая директория родительского процесса  
    &StartupInfo, // компоненты окна процесса  
    &ProcessInfo); // для получения информации о процессе
```

Первый параметр функции `::CreateProcess` в данном примере не содержит пути к файлу, следовательно, он должен находиться в текущей директории. Отсутствие второго параметра приведет к тому, что командная строка будет содержать только имя исполняемого файла. Дочерний процесс может использовать для доступа к командной строке вызов функции Win32 API `::GetCommandLine`, либо параметр `lpCmdLine`, передаваемый в функцию `WinMain`. Последний параметр передает адрес неинициализированной структуры типа `PROCESS_INFORMATION`. Функция записывает туда дескрипторы и идентификаторы дочернего процесса и его первичного потока.

Функция `CreateProcess` сразу возвращает управление. Дочерний процесс (как и любой другой) завершается либо по выходу из главной функции (для программ с графическим интерфейсом `WinMain`), либо вызовом функции `::ExitProcess` с передачей ей кода завершения. Родительский процесс может получить информацию о состоянии дочернего, вызвав функцию `::GetExitCodeProcess`.

```
DWORD ExitCode;
::GetExitCodeProcess (
    ProcessInfo.hProcess, // дескриптор процесса
    &ExitCode);           // адрес переменной для кода возврата
if (ExitCode == STILL_ACTIVE)
    // Процесс продолжает выполняться
else
    // Процесс завершен, в ExitCode содержится код возврата
```

Использование дескриптора процесса

Родительская программа может использовать функцию `::WaitForSingleObject`, передав ей дескриптор дочернего процесса для ожидания его завершения. Она также может использовать его для передачи другим функциям, например, `::SetPriorityClass` для изменения приоритета процесса или `::TerminateProcess` для его немедленного прекращения.

Дескриптор остается корректным даже после окончания процесса. Прекратить его использование (закрыть) можно, передав его функции `::CloseHandle`. Если дескриптор не был явно закрыт, он будет закрыт автоматически при завершении программы. Когда *все* дескрипторы процесса будут закрыты, Windows выгружает его из памяти.

Родительский и дочерний процесс изначально функционируют независимо. При завершении родительского процесса дочерние процессы *не завершаются автоматически*. Кроме того, *любой* процесс (а не только родительский) может управлять другим процессом, вызывая функции Win32 API (например, `::WaitForInputIdle`, `::GetExitCodeProcess`, `::WaitForSingleObject`, `::SetPriorityClass` или `::TerminateProcess`), при условии, что имеет его дескриптор.

Методы получения дескрипторов рассматриваются ниже.

19.2. Синхронизация процессов

Из четырех рассмотренных объектов синхронизации Win32 для синхронизации действий потоков внутри *разных процессов* можно использовать три: мьютексы, семафоры и события.

При использовании объекта синхронизации нужно передать его дескриптор соответствующей функции Win32 API. Однако дескриптор, полученный одним процессом (например, вызовом функции `::CreateMutex`), не может быть использован другим процессом. Вместо этого он должен получить собственный дескриптор этого объекта (если это не дескриптор, *унаследованный* дочерним процессом). Например, первый процесс создает мьютекс таким образом:

```
HMutex = ::CreateMutex  
(NULL,      // стандартные атрибуты защиты,  
            // дескриптор не может наследоваться  
FALSE,      // мьютекс изначально никому не принадлежит  
"Dummy Corporation Mutex"); // имя мьютекса
```

Тогда второй процесс получает дескриптор данного мьютекса, выполняя *точно такой же* оператор. Функция `CreateMutex` в этом случае не создает новый мьютекс, а возвращает новый дескриптор того же объекта, который может быть использован в данном процессе. При этом необходимо быть уверенным, что ни один из посторонних запущенных процессов не использует мьютекс с точно таким же именем.

Аналогичным образом несколько процессов могут совместно использовать один семафор или событие (соответственно функции `::CreateSemaphore` и `::CreateEvent`).

Альтернативный способ совместного использования дескрипторов – вызов функции Win32 API `::OpenMutex`, `::OpenSemaphore` или `::OpenEvent`. Например, один процесс создал мьютекс, как было показано выше. Другой процесс может получить к нему доступ следующим образом:

```
HMutex = ::OpenMutex  
(MUTEX_ALL_ACCESS, // разрешен максимальный доступ,  
FALSE,             // дескриптор не может наследоваться  
"Dummy Corporation Mutex"); // имя мьютекса
```

Если мьютекс с таким именем еще не был создан, то функция `OpenMutex` завершается неудачно и возвращает `NULL`. Таким образом эту функцию можно использовать, чтобы определить, создан мьютекс на данный момент или нет. Аналогичным образом используются функции `::OpenSemaphore` и `::OpenEvent`.

Получение дескриптора процесса

Процесс может также получить дескриптор другого процесса (даже если не он его создал), вызывая функцию Win32 API `::OpenProcess`.

В этом случае функции `OpenProcess` следует передать не имя (оно у процессов отсутствует), а *идентификатор* процесса:

```
HANDLE HProcess;  
DWORD IDProcess;  
// Получить идентификатор процесса и записать в IDProcess  
HProcess = ::OpenProcess  
    (PROCESS_ALL_ACCESS, // разрешен максимальный доступ,  
     FALSE,              // дескриптор не может наследоваться  
     IDProcess);         // идентификатор процесса
```

Хотя программа может заранее знать *имя* объекта, *идентификатор* объекта она должна получить *во время выполнения*.

Родительский процесс может получить идентификатор дочернего из поля `dwProcessId` структуры `LPPROCESS_INFORMATION`, а сам процесс может получить свой идентификатор, вызывая функцию Win32 API `::GetCurrentProcessId`.

Полученный дескриптор процесса можно использовать для вызова любой функции Win32 API, чтобы реализовать управление или ожидание завершения процесса.

Наследуемые дескрипторы

Существует еще два способа совместного использования процессами дескрипторов Windows: наследование и дублирование дескриптора.

Чтобы дочерний процесс мог *наследовать* дескриптор, при его создании родительский процесс должен объявить его наследуемым, например:

```
SECURITY_ATTRIBUTES Security =  
    {sizeof (SECURITY_ATTRIBUTES),  
     NULL,      // стандартные атрибуты защиты  
     TRUE};     // дескриптор МОЖЕТ НАСЛЕДОВАТЬСЯ  
HMutex = ::CreateMutex  
    (&Security, // задает атрибуты защиты  
     // и наследуемость дескриптора  
     FALSE,     // мьютекс изначально никому не принадлежит  
     NULL);     // мьютекс не имеет имени
```

Теперь при создании дочернего процесса функции `::CreateProcess` следует в качестве пятого параметра (`bInheritHandles`) передать значение `TRUE`. Это позволит дочернему процессу наследовать *любой* наследуемый дескриптор родительского процесса. Сам дескриптор передается любым способом передачи информации между процессами (рассматриваются далее).

Дублируемые дескрипторы

Если процесс имеет дескриптор некоторого объекта, то он может вызвать функцию Win32 API ::DuplicateHandle для генерирования нового дескриптора того же объекта, который может использоваться *любым* процессом, включая текущий, указанным соответствующим аргументом этой функции (через дескриптор). После этого дескриптор объекта передается процессу-приемнику одним из описанных ниже способов.

Наследование и дублирование дескрипторов не столь удобно, как первые два из описанных методов (::Create... и ::Open...), поскольку требуют организации передачи информации между процессами, однако применимы для использования широкого круга дескрипторов разных типов: объектов синхронизации, потоков, процессов, каналов, объектов файлов памяти и любых других, созданных функцией Win32 API ::CreateFile.

В Windows разным процессам *нельзя* совместно использовать дескрипторы следующих объектов, остающихся собственностью создавшего их процесса:

- выделенная память,
- графические объекты,
- окна.

19.3. Обмен данными между процессами

Обмен данными по каналам

Канал (pipe) – это специальный механизм передачи данных между процессами. Win32 предоставляет как *именованные* (кроме Windows 95), так и *анонимные* каналы.

Анонимные каналы обычно используются для связи между родительским и дочерним процессами. Сначала родительский процесс создает канал, вызывая функцию Win32 API ::CreatePipe. Та через аргументы возвращает два дескриптора: для записи и чтения. Их при вызове CreatePipe нужно сделать наследуемыми.

Перед созданием дочернего процесса эти дескрипторы можно сделать доступными для него вызовом функции Win32 API ::SetStdHandle. Эта функция модифицирует стандартные дескрипторы ввода, вывода и сообщений об ошибках. Если родительский процесс хочет передать данные дочернему, он должен использовать функцию SetStdHandle для присвоения дескриптора чтения канала стандартному дескриптору ввода, если получить данные, то присвоить дескриптор записи канала стандартному дескриптору вывода.

После этого родительский процесс создает канал, вызывая функцию CreatePipe. При этом он должен разрешить наследование дескриптора. При посылке данных родительский процесс использует функцию ::WriteFile, при получении – ::ReadFile. Эти же функции используются и

дочерним каналом. Для получения дескриптора чтения или записи канала используется функция Win32 API ::GetStdHandle.

Использование именованных каналов, а также механизма связи между процессами, называемого *почтовым ящиком (mailslot)*, предполагается рассмотреть в рамках учебного курса "Программирование в сетях Windows" и описать в соответствующем учебном пособии.

Совместное использование памяти

Каждый процесс выполняется в своем адресном пространстве, недоступном другому процессу. Однако использование механизма *файлов памяти* позволяет организовать доступ к общему блоку памяти для двух или более процессов. Этот механизм предоставляет процессам доступ к копии (*отображению*) файла в оперативной памяти, причем свопинг организуется операционной системой. Однако, если файлы памяти используются просто для передачи данных между процессами, не требуется создавать файл на диске.

Для выделения общего блока памяти и организации к нему доступа нужно сделать следующее. Сначала *каждый* процесс, участвующий в обмене, вызывает функцию Win32 API ::CreateFileMapping, как в следующем примере:

```
HANDLE HFileMapping;  
HFileMapping = ::CreateFileMapping  
    ((HANDLE) 0xFFFFFFFF, // новый файл отсутствует,  
                                     // используется системный файл  
    (LPSECURITY_ATTRIBUTES) NULL, // стандартная защита,  
                                     // дескриптор не наследуемый  
    PAGE_READWRITE,           // доступ для чтения и записи  
    0, // максимальный размер (64-битовый) задается двумя  
    1024, // 32-битовыми значениями. Здесь – 1 Кбт.  
    "MyFileMappingObject"); // имя объекта файла памяти
```

Первый (по времени) вызов функции ::CreateFileMapping создает *объект файла памяти* и возвращает его дескриптор. Второй (в другом процессе) просто возвращает дескриптор уже существующего объекта, действительный в рамках данного процесса. В примере эти вызовы ограничивают размер используемого блока памяти одним килобайтом.

Далее каждый процесс выделяет блок памяти, называемый *представлением файла*:

```
char* PtrSharedMemory;  
PtrSharedMemory = ::MapViewOfFile  
    (HFileMapping, // дескриптор объекта файла памяти  
    FILE_MAP_ALL_ACCESS, // доступ для чтения и записи  
    0, // смещение в общем блоке памяти (64-битовое),  
    0, // задается двумя 32-битовыми значениями. Здесь – 0.  
    1024); // отображать 1 Кбт
```

Первый (по времени) вызов функции `::MapViewOfFile` выделяет блок памяти и возвращает указатель на него. Второй (в другом процессе) возвращает указатель на тот же самый блок памяти. Поскольку теперь оба процесса работают с одной и той же областью памяти, возможно, им потребуется синхронизировать свою работу.

После завершения работы с общей областью памяти каждый из процессов сначала вызывает функцию Win32 API `::UnmapViewOfFile`, чтобы отменить представление файла, а затем `::CloseFile`, чтобы закрыть дескриптор объекта файла памяти.

19.4. Использование буфера обмена для передачи данных

Команды буфера обмена

Как правило, программа, которая использует буфер обмена, должна содержать в меню Edit команды Cut, Copy и Paste. Если для генерации исходного кода используется мастер AppWizard, он сам добавит эти команды в меню Edit. Их стандартные свойства перечислены в следующей таблице.

Идентификатор	Надпись	Интерактивная справка	Другие свойства
ID_EDIT_CUT	Cu&t\tCtrl+X	Cut the selection and put it on the Clipboard\nCut	—
ID_EDIT_COPY	&Copy\tCtrl+C	Copy the selection and put it on the Clipboard\nCopy	—
ID_EDIT_PASTE	&Paste\tCtrl+V	Insert Clipboard contents\nPaste	—

Следующая таблица описывает настройки акселераторов. При этом в меню отображаются только акселераторы по новому соглашению.

Команда	Идентификатор	Комбинация клавиш по старому соглашению	Комбинация клавиш по новому соглашению
Cut	ID_EDIT_CUT	Shift+Del	Ctrl+X
Copy	ID_EDIT_COPY	Ctrl+Ins	Ctrl+C
Paste	ID_EDIT_PASTE	Shift+Ins	Ctrl+V

Как правило, обработчики команд Cut, Copy и Paste добавляются с использованием мастера ClassWizard. Чаще их добавляют в класс представления. Для каждой из команд следует добавить обработчики COMMAND и UPDATE_COMMAND_UI. Обработчик сообщения UPDATE_COMMAND_UI вызывается в тот момент, когда меню Edit открывается впервые. В это момент можно отредактировать меню, в частности, сделать доступной или недоступной ту или иную команду. Обработчик COMMAND получает управление в момент выбора соответствующей команды.

Команда	Идентификатор	Тип сообщения	Обработчик сообщения
Cut	ID_EDIT_CUT	COMMAND	OnEditCut
Cut	ID_EDIT_CUT	UPDATE_COMMAND_UI	OnUpdateEditCut
Copy	ID_EDIT_COPY	COMMAND	OnEditCopy
Copy	ID_EDIT_COPY	UPDATE_COMMAND_UI	OnUpdateEditCopy
Paste	ID_EDIT_PASTE	COMMAND	OnEditPaste
Paste	ID_EDIT_PASTE	UPDATE_COMMAND_UI	OnUpdateEditPaste

Способ реализации обработчиков сообщений COMMAND и UPDATE_COMMAND_UI зависит от формата переносимых данных. Ниже мы рассматриваем этот вопрос.

Использование буфера обмена для переноса текста

Сначала мы рассмотрим вопрос об использовании буфера обмена для передачи простого текста, состоящего из печатаемых символов ANSI и не содержащего внедренных форматирующих кодов. Вопрос о работе с форматированным текстом будет рассмотрен в параграфе "Использование буфера обмена для передачи данных зарегистрированных форматов".

Отметим, что если класс представления наследуется от MFC-класса CEditView, то поддержка команд Cut, Copy и Paste в нем уже реализована. Достаточно только включить в меню соответствующие команды.

Собственная реализация команд Cut и Copy предполагает следующее. В классе представления заводится переменная (назовем ее m_IsSelection) типа BOOL, которой присваивается значение TRUE в случае, если в тексте выбран фрагмент. Тогда обработчики UPDATE_COMMAND_UI могут выглядеть так:

```
void CProgView::OnUpdateEditCut(CCmdUI* pCmdUI)
{
    // TODO: Добавьте собственный код обработчика
    pCmdUI->Enable (m_IsSelection);
}
```

```
void CProgView::OnUpdateEditCopy(CCmdUI* pCmdUI)
{
    // TODO: Добавьте собственный код обработчика
    pCmdUI->Enable (m_IsSelection);
}
```

Обработчики сообщений COMMAND для команд Cut и Copy должны добавлять в буфер обмена выделенный блок текста. Для этого необходимо выполнить следующие действия.

- Вызов функции Win32 API ::GlobalAlloc для выделения блока памяти, достаточного для размещения выделенного текста.
- Вызов функции Win32 API ::GlobalLock для блокирования блока памяти и получения его указателя.
- Копирование текста в выделенный блок памяти.
- Вызов функции Win32 API ::GlobalUnlock для освобождения выделенной памяти.
- Вызов функции CWnd::OpenClipboard для открытия буфера обмена (или глобальной функции ::OpenClipboard с передачей ей дескриптора окна).
- Вызов функции Win32 API ::EmptyClipboard для удаления текущего содержимого буфера обмена.
- Вызов функции Win32 API ::SetClipboardData для назначения буферу обмена выделенного блока памяти (через его дескриптор).
- Вызов функции Win32 API ::CloseClipboard для закрытия буфера.
- Если выполняется команда Cut, удаление выделенных данных из документа.

Текст, копируемый в буфер обмена посредством описанной процедуры, должен отвечать следующим требованиям:

- Текст состоит из печатаемых символов ANSI без внедренных форматирующих и управляющих кодов.
- Каждая строка заканчивается символами возврата каретки и перевода строки.
- Весь блок заканчивается нулевым символом.

Как видно из описанной процедуры, Windows сохраняет не данные, а дескриптор блока памяти. Программист должен сам позаботиться о сохранении своих данных и передаче дескриптора в буфер обмена. Поэтому первым нашим действием было размещение соответствующего блока памяти функцией Win32 API ::GlobalAlloc.

```
HGLOBAL GlobalAlloc(
    UINT uFlags,          // атрибуты размещения
    SIZE_T dwBytes );    // количество байтов
```

Функция возвращает дескриптор выделенного блока памяти (или NULL в случае невозможности выделения). При выделении блока памяти с целью последующей передачи в буфер обмена в первом параметре необходимо установить флаги GMEM_MOVEABLE и GMEM_DDESHARE. Если также установить флаг GMEM_ZEROINIT, блок памяти будет инициализирован нулями.

Ниже приводится пример кода, реализующего все описанные действия.

```
char Buffer[] = "Sample text to be copied to the Clipboard";
void CProgView::OnEditCopy()
{
    // TODO: Добавьте собственный код обработчика
    HGLOBAL HMem;
    char *PMem;
    // 1. Выделение блока памяти
    HMem = (char *) ::GlobalAlloc (GMEM_MOVEABLE|GMEM_DDESHARE,
                                   strlen (Buffer) + 1);

    if (HMem == NULL)
    {
        AfxMessageBox ("Error copying data to the Clipboard");
        return;
    }
    // 2. Блокирование блока памяти и получение указателя
    PMem = (char *) ::GlobalLock (HMem);
    if (PMem == NULL)
    {
        ::GlobalFree (HMem);
        AfxMessageBox ("Error copying data to the Clipboard");
        return;
    }
    // 3. Копирование выделенного текста в блок памяти
    ::lstrcpy (PMem, Buffer);
    // 4. Разблокирование выделенной памяти
    ::GlobalUnlock (HMem);
    // 5. Открыть буфер обмена
    if (!OpenClipboard ())
    {
        ::GlobalFree (HMem);
        AfxMessageBox ("Error copying data to the Clipboard");
        return;
    }
    // 6. Очистить буфер обмена
    ::EmptyClipboard ();
```

```
// 7. Передать дескриптор памяти в буфер обмена
::SetClipboardData (CF_TEXT, HMem);
HMem = 0; // Чтобы больше не использовать (это запрещено)
// 8. Закрыть буфер обмена
::CloseClipboard ();
}
```

Получение текста из буфера обмена

Если программа позволяет вставлять из буфера только стандартный текст, обработчик сообщения `UPDATE_COMMAND_UI` для команды `Paste` должен делать команду доступной, только если буфер содержит текст. Чтобы определить, содержит ли буфер обмена данные, соответствующие специальному формату, используется функция Win32 API `::IsClipboardFormatAvailable`. Ее первый аргумент задает желаемый формат. Пример вызова:

```
void CProgvew::OnUpdateEditPaste(CCmdUI* pCmdUI)
{
    // TODO: Добавьте собственный код обработчика
    pCmdUI->Enable (::IsClipboardFormatAvailable (CF_TEXT));
}
```

Одновременно в буфере обмена могут содержаться данные нескольких форматов. Чтобы определить все форматы, доступные в данный момент, используется функция Win32 API `::EnumClipboardFormats`. Чтобы определить лучший формат из предварительно заданного списка приоритетов, можно воспользоваться функцией Win32 API `::GetPriorityClipboardFormat`.

Обработчик сообщения `COMMAND` для команды `Paste` должен получать кусок текста из буфера обмена. Для этого необходимо выполнить следующие действия.

- Вызов функции `CWnd::OpenClipboard` для открытия буфера обмена.
- Вызов функции `Win32 API ::GetClipboardData` для получения дескриптора блока памяти, содержащего текст.
- Выделить временный буфер, чтобы хранить копию текста из буфера обмена.
- Вызов функции `Win32 API ::GlobalLock` для блокирования блока памяти буфера обмена и получения его указателя.
- Копирование текста во временный буфер.
- Вызов функции `Win32 API ::GlobalUnlock` для разблокирования блока памяти буфера обмена.
- Вызов функции `Win32 API ::CloseClipboard` для закрытия буфера.
- Обработать, если нужно, текст во временном буфере, вставить в документ и освободить буфер.

Ниже приводится пример кода, реализующего описанные действия. При этом предполагается, что функция `InsertText` вставляет данные из временного буфера в документ.

```
void CProgView::OnEditPaste()
{
    // TODO: Добавьте собственный код обработчика
    HANDLE HClipText;
    char *PClipText;
    char *PTempBuffer;
// 1. Открыть буфер обмена
    if (!OpenClipboard ())
    {
        AfxMessageBox ("Could not open Clipboard");
        return;
    }
// 2. Получение дескриптора данных буфера обмена
    HClipText = ::GetClipboardData (CF_TEXT);
    if (HClipText == NULL)
    {
        ::CloseClipboard ();
        AfxMessageBox ("Error obtaining text from Clipboard");
        return;
    }
// 3. Разместить временный буфер
    PTempBuffer = new char [::GlobalSize (HClipText)];
    if (PTempBuffer == NULL)
    {
        ::CloseClipboard ();
        AfxMessageBox ("Out of memory");
        return;
    }
// 4. Блокирование блока памяти и получение указателя
    PClipText = (char *) ::GlobalLock (HClipText);
    if (PClipText == NULL)
    {
        ::CloseClipboard ();
        delete [] PTempBuffer;
        AfxMessageBox ("Error obtaining text from Clipboard");
        return;
    }
// 5. Копирование текста из буфера
    ::lstrcpy (PTempBuffer, PClipText);
// 6. Разблокирование блока памяти буфера обмена
    ::GlobalUnlock (HClipText);
```

```
// 7. Закрыть буфер обмена
::CloseClipboard ();
// 8. Вставить текст в документ и очистить временный буфер
InsertText (PTempBuffer);
delete [] PTempBuffer;
}
```

Копирование растрового изображения в буфер обмена

Растровое изображение переносится в буфер обмена теми же командами Cut и Copy. Обработчики сообщения UPDATE_COMMAND_UI в этом случае ничем не отличаются от обработчиков в случае копирования текста.

Рассмотрим процедуру реализации обработчиков сообщения COMMAND в этом случае. Соответствующая функция должна выполнить следующие действия:

- Вызов функции CWnd::OpenClipboard для открытия буфера обмена (или глобальной функции ::OpenClipboard с передачей ей дескриптора окна).
 - Вызов функции Win32 API ::EmptyClipboard для удаления текущего содержимого буфера обмена.
 - Вызов функции Win32 API ::SetClipboardData, передавая ей значение CF_BITMAP, как первый параметр и дескриптор растрового изображения как второй. Ранее мы рассматривали способы создания растрового изображения и получения его дескриптора. В рассмотренном ниже примере создается пустое изображение и в него копируются данные.
 - Вызов функции Win32 API ::CloseClipboard для закрытия буфера.
- Если выполняется команда Cut, удаление выделенных графических данных из документа.

```
void CProgView::OnEditCopy()
{
    // TODO: Добавьте собственный код обработчика
    CBitmap BitmapClip;
    CClientDC ClientDC(this);
    CDC MemDC;
    RECT Rect;
    // Создание пустого растрового изображения (для примера)
    GetClientRect (&Rect);
    BitmapClip.CreateCompatibleBitmap(&ClientDC,
        Rect.right - Rect.left, Rect.bottom - Rect.top);
    // Создание объекта памяти и копирование в него изображения
    MemDC.CreateCompatibleDC (&ClientDC);
    MemDC.SelectObject (&BitmapClip);
}
```



```

MemDC.BitBlt (0, 0, Rect.right - Rect.left,
              Rect.bottom - Rect.top,
              &ClientDC, 0, 0, SRCCOPY);
// 1. Открыть буфер обмена
if (!OpenClipboard ())
    return;
// 2. Очистить буфер обмена
::EmptyClipboard ();
// 3. Передать дескриптор изображения в буфер обмена
::SetClipboardData (CF_BITMAP, BitmapClip.m_hObject);
// предотвратить удаление растрового изображения
BitmapClip.Detach ();
// 4. Закрыть буфер обмена
::CloseClipboard ();
}

```

Комментарии.

При вызове функции `::SetClipboardData` в качестве первого параметра указывается значение `CF_BITMAP`, соответствующее формату растрового изображения, а в качестве второго – `BitmapClip.m_hObject`. Переменная `m_hObject` наследуется классом `CBitmap` от класса `CGdiObject` и содержит дескриптор растрового изображения.

После вызова функции `::SetClipboardData` для помещения изображения (точнее его дескриптора) в буфер обмена, использовать и удалять его не нужно. Вызов функции `Detach` класса `CGdiObject` предотвращает автоматическое удаление растрового изображения при удалении `BitmapClip` при выходе из функции `OnEditCopy`.

Получение растрового изображения из буфера обмена

Обработчик сообщения `UPDATE_COMMAND_UI` для команды `Paste` аналогичен обработчику в случае работы с текстом. Разница только в значении, которое получает функция `::IsClipboardFormatAvailable`:

```

void CProgView::OnUpdateEditPaste(CCmdUI* pCmdUI)
{
    // TODO: Добавьте собственный код обработчика
    pCmdUI->Enable(::IsClipboardFormatAvailable (CF_BITMAP));
}

```

Для реализации обработчика сообщения `COMMAND` нужно выполнить следующие действия:

- Вызов функции `CWnd::OpenClipboard` для открытия буфера обмена.
- Вызов функции Win32 API `::GetClipboardData`, передавая ей значение `CF_BITMAP` для получения дескриптора растрового изображения.

- Использование полученного дескриптора для копирования или отображения изображения (без его изменения).
- Вызов функции Win32 API ::CloseClipboard для закрытия буфера.

В приводимом ниже примере функция-обработчик получает изображение и отображает его в окне представления (в левом верхнем углу).

```
void CProgView::OnEditPaste()
{
    // TODO: Добавьте собственный код обработчика
    CBitmap BitmapClip;
    CClientDC ClientDC(this);
    BITMAP BitmapClipInfo;
    HANDLE HBitmapClip;
    CDC MemDC;
    // 1. Открыть буфер обмена
    if (!OpenClipboard ())
        return;
    // 2. Получение дескриптора данных из буфера обмена
    HBitmapClip = ::GetClipboardData (CF_BITMAP);
    if (HBitmapClip == NULL)
    {
        ::CloseClipboard ();
        return;
    }
    // 3. Использование дескриптора данных для их отображения
    // Инициализировать объект растрового изображения
    BitmapClip.Attach (HBitmapClip);
    // Получить информацию о растровом изображении
    BitmapClip.GetObject (sizeof(BITMAP), &BitmapClipInfo);
    // Создать объекта контекста устройства и
    // скопировать изображение в рабочую область
    MemDC.CreateCompatibleDC (&ClientDC);
    MemDC.SelectObject (&BitmapClip);
    ClientDC.BitBlt (0, 0, BitmapClipInfo.bmWidth,
        BitmapClipInfo.bmHeight, &MemDC, 0, 0, SRCCOPY);
    // Удалить дескриптор из объекта растрового изображения
    BitmapClip.Detach ();
    // 4. Закрыть буфер обмена
    ::CloseClipboard ();
}
```

Замечание. Если программа предусматривает передачу и текста, и изображений, то функция GetClipboardData должна проверять оба формата (CF_TEXT и CF_BITMAP), а затем выполнять ту часть кода, которая соответствует обнаруженному типу данных.

Использование буфера обмена для передачи данных зарегистрированных форматов

Стандартные форматы данных перечислены в документации на функцию `::SetClipboardData`. Если требуется использовать буфер обмена для передачи данных в собственном формате, например текст с внедренной информацией о форматировании, следует этот формат зарегистрировать вызовом функции Win32 API `::RegisterClipboardFormat`. Например:

```
UINT TextFormat;  
// ...  
TextFormat = ::RegisterClipboardFormat("MyTextFormat");
```

Добавляя в буфер обмена текст в данном формате, следует в качестве первого параметра функции `SetClipboardData` указывать не стандартное значение, а зарегистрированный тип `TextFormat`:

```
::SetClipboardData (TextFormat, HMyText);
```

Здесь предполагается, что `HMyText` задает дескриптор блока памяти, выделенный функцией `::GlobalAlloc`. Точно так же тип `TextFormat` следует использовать при вызове функций `::IsClipboardFormatAvailable` и `::GetClipboardData`.

Если идентификатор формата уже зарегистрирован, функция `RegisterClipboardFormat` не регистрирует новый формат, а возвращает идентификатор этого же формата, зарегистрированного ранее. Это позволяет различным процессам обмениваться данными в собственном формате, следует лишь позаботиться об уникальности имени, передаваемого функции `RegisterClipboardFormat`.

В буфер обмена можно добавить несколько блоков данных, если они имеют разные форматы. Это позволит программам, не знающим ничего о вашем формате, также принимать информацию из буфера обмена. Например, вместе с форматированным текстом рекомендуется вставить в буфер обмена и неформатированный вариант:

```
::SetClipboardData (TextFormat, HMyText);  
::SetClipboardData (CF_TEXT, HPlainText);
```

Здесь предполагается, что значение `HPlainText` задает дескриптор блока памяти, содержащего неформатированный текст. Для того, чтобы узнать все форматы данных, содержащихся в буфере обмена, используется, как было отмечено ранее, функция `::EnumClipboardFormats`. Она возвращает типы форматов в том порядке, в котором они добавлялись в буфер.

Тема 20. Механизм OLE

Рассмотренные выше способы обмена информацией предполагали, что программа, принимающая данные, полностью понимает их формат и при необходимости может осуществлять обработку этих данных.

Механизм OLE – Object Linking and Embedding (*связывание и внедрение объектов*) – метод обмена информацией между программами, позволяющей программе, принимающей данные, не воспринимать их формат. Вместо этого она устанавливает связь с программой, создавшей эти данные. С этого момента за их отображение и редактирование отвечает исходная программа и механизм OLE. Таким образом, можно передавать данные *любого* формата *любой* программе, поддерживающей этот механизм.

Таким образом можно создать документ, содержащий блоки данных, созданных различными программами – *составной* (*compound*) документ.

В данной теме мы рассмотрим только основные моменты механизма OLE, ограничившись базовыми возможностями, предоставляемыми библиотекой MFC и мастерами Visual C++.

А именно будут рассмотрены три основных механизма OLE: внедрение, связывание и автоматизация. В качестве примеров будут созданы простейшие варианты *сервера* OLE (исходная программа) и *контейнера* OLE (программа, принимающая данные OLE).

20.1. Внедрение, связывание и автоматизация

Наиболее распространенным из этих трех механизмов является *внедрение объекта*. Под объектом здесь понимается блок данных, созданный сервером и отображаемый в контейнере OLE. Приложение-контейнер сохраняет его как часть документа контейнера, в который вставлен дополнительный компонент данных.

Первый способ внедрения – использование буфера обмена для копирования блока данных из сервера в контейнер. При этом, если данные находятся не в собственном формате данных контейнера, то последний не просто передает данные из буфера в документ, а использует механизм внедрения. Контейнер использует *отдельную копию* данных, так что изменение оригинала сервером на данных контейнера не отражается. Обычно для явного внедрения объекта меню Edit содержит дополнительно команду Paste Special.

Второй способ внедрения предполагает наличие команды Insert New Object ... в меню Edit программы-контейнера. При ее выборе должно отобразиться диалоговое окно со списком типов объектов. Каждая установленная программа-сервер OLE регистрирует в реестре Windows один или несколько типов объектов – они и перечисляются в упомянутом окне. Собственно отличие от первого способа состоит в том, что сервер

запускается не для копирования существующего блока данных, а для создания нового.

Редактирование внедренного объекта также происходит двумя способами.

Первый способ – редактирование *на месте (in place)*. Объект отображается в окне контейнера. Однако программа-сервер временно объединяет свои команды меню и кнопки панели инструментов с аналогичными средствами программы-контейнера. Она также делает доступными свои комбинации клавиш. Чтобы инициировать этот способ редактирования, нужно выделить объект и выбрать команду Edit в подменю Object меню Edit контейнера. Подменю Object озаглавливается в соответствии с типом внедренного объекта.

Второй способ – редактирование внедренного объекта в окне программы-сервера. Такой режим называют *полностью открытым (fully opened)*. Сервер открывает собственное окно, не используя окно контейнера. Чтобы инициировать этот способ редактирования, нужно выделить объект и выбрать команду Open в подменю Object меню Edit контейнера.

Основной особенностью механизма *связывания* является то, что данные объекта сохраняются в программе-сервере, как один из ее документов. При этом данные могут составлять часть или весь документ сервера. Контейнер связан с данными исходного документа и рассматривает их как часть своего документа, однако сам их не хранит.

Чтобы внедрить связанный объект, обычно в контейнере предусматривается команда Paste Link в меню Edit, либо опция Paste Link при выполнении команды Paste Special... меню Edit. Для редактирования после выделения объекта (двойным щелчком) обычно выбирают команду Open или Edit в подменю Object меню Edit контейнера. Другой способ – непосредственный запуск программы-сервера и открытие исходного документа. В любом случае редактирование выполняется в окне сервера, т.е. в полностью открытом режиме.

Третий механизм, поддерживаемый OLE, называется *автоматизацией*. При его реализации сервер автоматизации использует некоторые из своих средств совместно с другой программой – клиентом автоматизации. Например, программа-браузер может предложить свои средства передачи данных другим программам, которые могут их использовать, скажем, для отображения Web-страниц или загрузки файлов по FTP-протоколу. Клиент при этом может изменять некоторые данные сервера (*свойства*) или вызывать некоторые его функции (*методы*).

Программа клиента автоматизации может содержать некоторый макроязык, позволяющий управлять другими программами, например, можно написать макрос для управления программами Microsoft Office.

Библиотека MFC поддерживает сервер и клиента автоматизации, а мастера Visual C++ генерируют исходный текст программ автоматизации.

20.2. Разработка программы-сервера

Здесь в качестве примера мы создадим простой OLE-сервер ServDemo. В качестве основы мы возьмем одну из прежних версий программы MiniDraw, позволявшую создавать рисунки из прямых линий и сохранять их на диске.

Будут реализованы оба метода редактирования внедренных объектов: на месте и в полностью открытом режиме. Программа ServDemo не поддерживает связывания, объекты внедряются только командой New Object... меню Edit (не поддерживается команды вставки, копирования и технология drag-and-drop).

Генерация исходного текста программы

Как обычно, для этой цели воспользуемся мастером AppWizard. Назовем проект ServDemo, при генерации файлов продедаем те же шаги, что и для программы WinGreet, сделав следующие изменения:

- В диалоговом окне Step 3 выберем опцию Full-server, отменим метку выбора опции ActiveX Control, прочие опции оставим неизменными;
- В диалоговом окне Step 4 щелкнем на кнопке Advanced... и во вкладке Document Template Strings в поле File extension введем srv. Это стандартное расширение для файлов, сохраняемых программой.

В результате будет сгенерирован код программы, которую можно запустить и как автономное приложение, и как сервер OLE, поддерживающий внедренные и связанные компоненты (хотя связывание мы реализовывать не будем). Обратите внимание, что мастер создал два новых класса: класс компонента сервера CServDemoSrvrItem и класс редактирования CInPlaceFrame.

Класс приложения

Для поддержки OLE мастер AppWizard включает в предварительно компилируемый файл заголовков StdAfx.h подключение файла AfxOLE.h. Вносятся дополнения и в стандартную функцию InitInstance класса приложения. Во-первых, это вызов глобальной MFC-функции AfxOleInit для инициализации поддержки механизма OLE:

```
if (!AfxOleInit())
{
    AfxMessageBox(IDP_OLE_INIT_FAILED);
    return FALSE;
}
```

После создания шаблона документа добавляется вызов функции SetServerInfo класса CSingleDocTemplate:

```
pDocTemplate->SetServerInfo(
    IDR_SRVR_EMBEDDED, IDR_SRVR_INPLACE,
    RUNTIME_CLASS(CInPlaceFrame));
```

Здесь определяются идентификаторы меню, отображаемого программой ServDemo при редактировании в режиме полного открытия (IDR_SRVR_EMBEDDED), и меню, отображаемого при редактировании "in place" (IDR_SRVR_INPLACE). Они определяют соответствующие ресурсы комбинаций клавиш. Также определяется класс управления окном, которое обрамляет компонент OLE при редактировании на месте (класс CInPlaceFrame).

Мастер AppWizard добавляет определение новой переменной (m_server) в класс приложения. Она является экземпляром класса COleTemplateServer и называется *объектом шаблона сервера*. Этот объект при запуске программы как сервера OLE создает новый объект документа, используя информацию, хранящуюся в шаблоне документа. AppWizard добавляет в функцию InitInstance вызов

```
m_server.ConnectTemplate(clsid, pDocTemplate, TRUE);
```

Здесь первый параметр задает уникальный идентификатор типа документа, который был сгенерирован мастером:

```
// {03A2EC83-ABBE-11D1-80FC-00C0F6A83B7F}
static const CLSID clsid =
    { 0x3a2ec83, 0xabbe, 0x11d1,
      { 0x80, 0xfc, 0x0, 0xc0, 0xf6, 0xa8, 0x3b, 0x7f } };
```

Второй параметр является указателем на шаблон документа. Значение TRUE, переданное в качестве третьего параметра, означает, что при вызове внедренного объекта контейнером сервера для редактирования каждый раз запускается новый экземпляр сервера. В SDI-приложениях необходимо передавать именно это значение, так как они могут управлять только одним документом.

При выбранной опции Full-server мастер AppWizard добавляет вызовы функций EnableShellOpen и RegisterShellFileTypes для регистрации типа файла (см. тему "Хранение данных"):

```
EnableShellOpen();
RegisterShellFileTypes(TRUE);
```

Если программа запускается как OLE-сервер, то требуется вызвать функцию COleTemplateServer::RegisterAll для ее регистрации как сервера с библиотеками OLE, что необходимо для редактирования внедренного компонента. В этом случае программа не показывает главное окно и не вызывает функции создания или открытия документа (нет вызова ProcessShellCommand):

```

// Запущена ли программа как сервер OLE
if (cmdInfo.m_bRunEmbedded || cmdInfo.m_bRunAutomated)
{
// Регистрируйте все серверы OLE как исполнимые, чтобы
// сделать библиотеки OLE доступными из других приложений.
    COleTemplateServer::RegisterAll();
// Приложение запущено с ключом /Embedding или /Automation.
// В этом случае не показывайте главное окно
    return TRUE;
}

```

Далее мастер AppWizard добавил вызов функции COleTemplateServer::UpdateRegistry:

```

// В случае автономного запуска приложения-сервера
// восстановим системный реестр (вдруг был поврежден)
m_server.UpdateRegistry(OAT_INPLACE_SERVER);

```

Этот вызов вводит в системный реестр Windows информацию о программе и типе внедренного компонента, который он поддерживает.

Регистрация возможна одним из следующих способов:

- Запуск сервера как автономной программы (происходит вызов UpdateRegistry).
- Двойным щелчком на файле ServDemo.reg (создается мастером AppWizard). В этом случае запускается программа RegEdit.exe, которая и вводит информацию в системный реестр.

И наконец, последнее, что сделал AppWizard при генерации файлов (при выбранной опции Full-server) – это вызов функции DragAcceptFiles для поддержки технологии drag-and-drop.

```
m_pMainWnd->DragAcceptFiles();
```

Впрочем, это изменение к технологии OLE отношения не имеет.

Класс документа

В нашем случае класс документа программы порождается не от класса CDocument, а от класса COleServerDoc, дополнительно предоставляющего средства управления документом при выполнении программы как сервера OLE.

Кроме того, класс документа переопределяет виртуальную функцию OnGetEmbeddedItem, которая вызывается при запуске программы как сервера OLE. Функция создает объект типа CServDemoSrvrItem, который будет использоваться в дальнейшем.

Необходимый код генерируется мастером:


```

COleServerItem* CServDemoDoc::OnGetEmbeddedItem()
{
    // OnGetEmbeddedItem вызывается системой для получения
    // объекта CServDemoSrvrItem, связанного с документом.
    // Вызывается только при необходимости.

    CServDemoSrvrItem* pItem = new CServDemoSrvrItem(this);
    ASSERT_VALID(pItem);
    return pItem;
}

```

Класс компонента сервера

При выборе опции Full-server AppWizard создает новый класс CServDemoSrvrItem, производный от MFC-класса COleServerItem. Он предоставляет дополнительные средства обработки документа, предназначенные для создания и редактирования внедренного компонента. Объект этого класса, как мы видели, создается при вызове функции OnGetEmbeddedItem. Файлы определения и реализации класса создаются мастером и имеют имена SrvrItem.h и SrvrItem.cpp соответственно.

Функция CServDemoSrvrItem::OnDraw вызывается, когда программа-контейнер отображает внедренный объект в своем окне представления (в случае, если компонент *неактивен*, т.е. не редактируется сервером). В режиме полного открытия он отображается в окне сервера.

Вместо отображения в окне контейнера функция CServDemoSrvrItem::OnDraw генерирует метафайл, содержащий команды, необходимые для отображения компонента. Этот файл *проигрывается* в окне представления контейнера. Созданная мастером реализация лишь частично выполняет эти функции, поэтому ниже исходный код будет модифицирован и дополнен.

Функция OnDraw класса представления вызывается для отображения документа при автономном выполнении программы (компонент *активен*).

Функция CServDemoSrvrItem::OnGetExtent возвращает размер внедренного компонента OLE, когда контейнер запрашивает его размер. Мастер генерирует стандартную реализацию этой функции, возвращающую размер 3000*3000 единиц HIMETRIC (равных 0.01 мм).

Класс окна редактирования на месте

Мастер AppWizard создает класс CInPlaceFrame, производный от MFC-класса COleIPFrameWnd. Когда программа-сервер выполняется автономно или в режиме полного открытия, окно представления обычно совпадает с главным окном (экземпляр класса CFrameWnd). В случае редактирования "на месте" окно представления ограничивается специальным обрамляющим окном редактирования, управляемым экземпляром класса

CInPlaceFrame, который определен и реализован в созданных мастером файлах IpFrame.h и IpFrame.cpp.

Мастер AppWizard определяет в классе CInPlaceFrame переменную m_wndResizeBar – экземпляр класса COleResizeBar. Она отвечает за изменение размеров окна редактирования "на месте". Создание соответствующего объекта (вызов функции COleResizeBar::OnCreate) мастер генерирует в коде реализации функции CInPlaceFrame::OnCreate.

Обрамляющее окно редактирования "на месте" отображается только в процессе редактирования внедренного компонента сервером. Если компонент неактивен, то программа-контейнер обычно отображает его границу самостоятельно. Она может позволять изменять его размеры и иметь соответствующие маркеры. В примере, рассмотренном ниже, такие возможности не предусмотрены.

Класс представления

В класс представления программы мастер AppWizard добавляет функцию OnCancelEditSrvr, которая получает управление после нажатия клавиши Esc при редактировании "на месте". Она в свою очередь вызывает функцию OnDeactivateUI класса COleServerDoc, завершающую сеанс редактирования в этом режиме:

```
void CServDemoView::OnCancelEditSrvr()
{
    GetDocument() ->OnDeactivateUI(FALSE);
}
```

Клавиша Esc определена с идентификатором ID_CANCEL_EDIT_SRVR в таблице акселераторов IDR_SRVR_INPLACE. Функция OnCancelEditSrvr также добавляется мастером в схему сообщений для ее вызова в ответ на использование акселератора.

Ресурсы

В файл ресурсов программы мастер AppWizard включает файл Afxholesv.rc, определяющий некоторые ресурсы, используемые MFC-классами сервера OLE.

Для каждого из трех возможных режимов работы мастер определяет свое меню и таблицу акселераторов. Режимам соответствуют идентификаторы ресурсов IDR_MAINFRAME, IDR_SRVR_EMBEDDED и IDR_SRVR_INPLACE. Далее мы внесем в эти ресурсы необходимые модификации.

Добавление кода, реализующего основные функции

Программа ServDemo предназначена для создания простых рисунков, состоящих из отрезков прямых линий. Чтобы заново не писать весь код, реализующий эту функциональность, можно просто скопировать в сгенерированный исходный код текст соответствующих функций из версии

программы MiniDraw, разработанной при изучении темы "Хранение данных". При этом не следует забывать, что имена основных классов у нас изменились. Потребуется сделать следующее.

В классе представления:

- Добавить отсутствующие переменные и их инициализацию в конструкторе.
- Добавить с помощью ClassWizard обработчики событий OnLButtonDown, OnLButtonUp и OnMouseMove.
- Скопировать код перечисленных функций-обработчиков, а также код функций OnDraw и PreCreateWindow.

В классе документа:

- Скопировать объявление класса CLine и код его реализации.
- Добавить переменную m_LineArray, а также объявление функций AddLine, GetLine и GetNumLines.
- Добавить с помощью ClassWizard обработчики событий OnEditClearAll, OnUpdateEditClearAll, OnEditUndo и OnUpdateEditUndo.
- Скопировать код перечисленных функций, а также код функций OnDraw, PreCreateWindow, Serialize и DeleteContents.

Добавление поддержки OLE – меню

Прежде всего, настроим меню. Мастер AppWizard создал в программе три меню:

- IDR_MAINFRAME. Отображается при автономном выполнении.
- IDR_SRVR_EMBEDDED. Отображается при выполнении как сервера OLE в открытом режиме.
- IDR_SRVR_INPLACE. Отображается при выполнении как сервера OLE при редактировании на месте. В нем отсутствует меню File. При редактировании на месте система *объединяет* меню сервера и контейнера, чтобы обе программы отображали одинаковые меню (они реализуются как всплывающие).

Меню File и Help мы изменять не будем. Отметим только, что при выполнении программы как сервера OLE в открытом режиме текст заголовков команд Update и Exit будет изменяться (добавляется имя документа).

Меню Edit во всех трех вариантах мы отредактируем так, чтобы оно содержало только следующие три элемента:

Идентификатор	Надпись	Интерактивная справка	Другие свойства
ID_EDIT_UNDO	&Undo\tCtrl+Z	Undo the last action\nUndo	—
—	—	—	Separator
ID_EDIT_CLEAR_ALL	&Delete All	Erase everything\nErase All	—

Как было отмечено выше, при редактировании на месте меню сервера и контейнера объединяются. Результирующее объединенное меню содержит следующие всплывающие меню, размещенные слева направо:

- Все всплывающие меню в меню контейнера для редактирования на месте, расположенные перед двойным разделителем. В примере программы контейнера, рассматриваемом ниже, это будет меню File.
- Все всплывающие меню в меню сервера для редактирования на месте, расположенные перед двойным разделителем. В нашем случае это меню Edit.
- Оставшиеся всплывающие меню в меню контейнера для редактирования на месте, если они есть. В примере программы контейнера, рассматриваемом ниже, таких меню не будет.
- Оставшиеся всплывающие меню в меню сервера для редактирования на месте, если они есть. В нашем случае это меню Help.

Добавление поддержки OLE – исходный код

Добавим в раздел `public` определения класса `CLine` (файл `ServDemoDoc.h`) объявления двух функций:

```
class CLine : public CObject
{
// ...
void Draw (CDC *PDC);
int GetMaxX () { return m_X1 > m_X2 ? m_X1 : m_X2; }
int GetMaxY () { return m_Y1 > m_Y2 ? m_Y1 : m_Y2; }
virtual void Serialize (CArchive& ar);
};
```

В том же файле добавим в раздел `public` определения класса `CServDemoDoc` объявление функции `GetDocSize`:

```
class CServDemoDoc : public COleServerDoc
{
// ...
public:
void AddLine (int X1, int Y1, int X2, int Y2);
CSize GetDocSize ();
// ...
};
```

В файле `ServDemoDoc.cpp` после определения функции `AddLine` добавим реализацию метода `GetDocSize`:

```
CSize CServDemoDoc::GetDocSize ()
{
int XMax = 1, YMax = 1;
int X, Y;
```

```

int Index = m_LineArray.GetSize ();
while (Index--)
{
    X = m_LineArray.GetAt (Index)->GetMaxX ();
    XMax = X > XMax ? X : XMax;
    Y = m_LineArray.GetAt (Index)->GetMaxY ();
    YMax = Y > YMax ? Y : YMax;
}
return CSize (XMax, YMax);
}

```

Функция GetDocSize вычисляет текущий размер рисунка путем нахождения максимальных значений координат образующих его линий.

В реализации класса представления необходимо сделать так, чтобы функция COleServerDoc::UpdateAllItems вызывалась при каждом изменении содержимого документа или размера окна представления. Этот вызов заставляет OLE вызвать функцию OnDraw класса CServDemoSrvrItem для повторного создания метафайла, позволяющего отобразить внедренный компонент в окне контейнера. В нашем случае соответствующий вызов достаточно добавить в код функций OnDraw и OnLButtonUp (файл ServDemoView.cpp):

```

void CServDemoView::OnDraw(CDC* pDC)
{
    // ...
    pDoc->UpdateAllItems (0);
}

void CServDemoView::OnLButtonUp(UINT nFlags, CPoint point)
{
    if (m_Dragging)
    {
        // ...
        PDoc->UpdateAllItems (0);
    }
    CView::OnLButtonUp(nFlags, point);
}

```

Добавим код в функцию OnDraw класса CServDemoSrvrItem (файл SrvrItem.cpp):

```

BOOL CServDemoSrvrItem::OnDraw (CDC* pDC, CSize& rSize)
{
    // Удалить это, если используется параметр rSize
    UNREFERENCED_PARAMETER(rSize);

    CServDemoDoc* pDoc = GetDocument ();
    ASSERT_VALID (pDoc);
}

```

```

// TODO: установите режим отображения и размер
pDC->SetMapMode (MM_ANISOTROPIC);
pDC->SetWindowOrg (0,0);

// Заменить сгенерированный AppWizard вызов SetWindowExt:
pDC->SetWindowExt (pDoc->GetDocSize ());

// TODO: добавьте код отображения. Все рисунки
// помещаются в метафайл контекста устройства (pDC)
// Такой же код, как и в CServDemoView::OnDraw:
int Index = pDoc->GetNumLines ();
while (Index--)
    pDoc->GetLine (Index)->Draw (pDC);
return TRUE;
}

```

20.3. Разработка программы-контейнера

Созданная программа-сервер может функционировать как автономное приложение. Однако для использования ее как сервера OLE необходимо создать программу-контейнер. Для этой цели воспользуемся мастером AppWizard. Назовем проект ContDemo, при генерации файлов сделаем те же шаги, что и для программы WinGreet, за исключением того, что на этапе Step 3 выберем опцию Container.

Класс приложения

Как и в случае программы-сервера, AppWizard добавил в код функции InitInstance класса приложения (файл ContDemo.cpp) инициализацию библиотеки OLE:

```

if (!AfxOleInit())
{
    AfxMessageBox(IDP_OLE_INIT_FAILED);
    return FALSE;
}

```

Он также добавил вызов функции CSingleDocTemplate::SetContainerInfo после создания шаблона документа:

```

pDocTemplate->SetContainerInfo(IDR_CNTR_INPLACE);

```

Этот вызов задает идентификатор меню и акселератора, которые используются при редактировании на месте. Как мы отмечали раньше, это меню объединяется с соответствующим меню сервера.

Класс документа

Класс документа CContDemoDoc порождается не от класса CDocument, а от класса COleDocument, содержащего основные средства для управления документами в контейнере OLE.

Объект документа в контейнере OLE управляет сохранением внедренного компонента или компонентов, а также собственно данными документа. В нашем варианте программа ContDemo сохраняет только внедренные данные. Они сохраняются в объекте класса CContDemoCntrlItem, производного от COleClientItem и управляемого классом документа.

Мастер AppWizard добавляет в метод Serialize в реализации класса документа (файл ContDemoDoc.cpp) вызов функции COleDocument::Serialize для сериализации внедренных компонентов. Функция Serialize вызывается для каждого объекта класса CContDemoCntrlItem, сохраняющего внедренный документ.

```
void CContDemoDoc::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
    {
        // TODO: добавьте код сохранения
    }
    else
    {
        // TODO: добавьте код загрузки
    }
    // Вызов функции базового класса для сериализации
    // объектов класса COleClientItem документа контейнера
    COleDocument::Serialize(ar);
}
```

Класс компонента контейнера

Мастер AppWizard создает новый класс CContDemoCntrlItem, порождаемый от MFC-класса COleClientItem и реализованный в файлах CntrlItem.h и CntrlItem.cpp. Когда новый компонент OLE внедряется в программу, класс представления создает объект класса CContDemoCntrlItem для хранения компонента и управления им.

AppWizard переопределяет некоторые виртуальные функции класса COleClientItem. Так, виртуальная функция CContDemoCntrlItem::OnChange получает управление при изменении компонента сервером в режиме редактирования на месте или в режиме полного открытия. Версия, сгенерированная мастером, вызывает версию этой функции в базовом классе, а затем вызывает функцию UpdateAllViews класса CDocument, после чего функция OnDraw класса представления перерисовывает компонент.

```

void CContDemoCntrlItem::OnChange
    (OLE_NOTIFICATION nCode, DWORD dwParam)
{
    ASSERT_VALID(this);
    COleClientItem::OnChange(nCode, dwParam);
    GetDocument() ->UpdateAllViews(NULL);
}

```

Когда внедренный компонент редактируется на месте, OLE вызывает виртуальную функцию `CContDemoCntrlItem::OnGetItemPosition`, чтобы получить размер и позицию компонента. Функция возвращает их в единицах устройства относительно области окна представления контейнера. Ее версия в базовом классе ничего не делает. В коде, сгенерированном мастером, она возвращает постоянный размер и позицию.

```

void CContDemoCntrlItem::OnGetItemPosition(CRect& rPosition)
{
    ASSERT_VALID(this);
    // TODO: возвратите правильный прямоугольник в rPosition
    rPosition.SetRect(10, 10, 210, 210);
}

```

Наконец, функция `OnDeactivateUI` вызывается, когда пользователь нажимает Esc для того, чтобы вывести внедренный компонент из активного состояния. Ее версия, сгенерированная мастером, вызывает функцию `OnDeactivateUI` базового класса для деактивизации внедренного компонента и освобождения занятых им ресурсов.

```

void CContDemoCntrlItem::OnDeactivateUI(BOOL bUndoable)
{
    COleClientItem::OnDeactivateUI(bUndoable);
    // Скрыть объект, если это не внедренный внешний объект
    DWORD dwMisc = 0;
    m_lpObject->GetMiscStatus(GetDrawAspect(), &dwMisc);
    if (dwMisc & OLEMISC_INSIDEOUT)
        DoVerb(OLEIVERB_HIDE, NULL);
}

```

Класс представления

Мастер AppWizard объявляет новую переменную `m_pSelection` класса представления `CContDemoView`:

```
CContDemoCntrlItem* m_pSelection;
```

Она инициализируется нулевым значением в функции `OnInitialUpdate` класса `CContDemoView`. При внедрении компонента OLE в нее записывается адрес объекта класса `CContDemoCntrlItem`, управляющего этим компонентом. Это делает функция `CContDemoView::OnInsertObject`. В нашей

программе она всегда содержит указатель на последний внедренный объект или NULL, если такового нет. В более серьезных программах обычно она используется для хранения адреса *выбранного* компонента.

Функция `CContDemoView::OnDraw`, сгенерированная мастером, отображает последний внедренный компонент, передавая значение `m_pSelection` функции `COleClientItem::Draw`. Обычно в программе-контейнере функция `OnDraw` класса представления рисует все внедренные объекты, а также собственно данные контейнера.

Мастер AppWizard также добавляет в класс представления обработчик команды `New Object...` из меню `Edit` программы-контейнера. Он называется `OnInsertObject` и делает следующее (см. файл `ContDemoView.cpp`):

- Отображает диалоговое окно `Insert Object`, позволяющее выбрать тип внедренного компонента.
- После закрытия диалогового окна создает объект `CContDemoCntrlItem` для управления внедренным компонентом. При этом конструктору класса передается адрес документа программы.
- Инициализирует объект класса `CContDemoCntrlItem`, используя информацию объекта диалогового окна типа `COleInsertDialog`.
- Активирует компонент для его редактирования на месте.
- Присваивает переменной `m_pSelection` адрес объекта `CContDemoCntrlItem` для нового внедренного компонента.

Мастер AppWizard также добавляет в класс представления обработчик команды для клавиши `Esc`, являющийся одной из клавиш-акселераторов `IDR_CNTR_INPLACE`, действующих при редактировании на месте. Обработчик называется `OnCancelEditCntr` и вызывает функцию `COleClientItem::Close` для закрытия компонента. Эта функция работает в сочетании с функцией `OnCancelEditSrvr` класса представления сервера.

Ресурсы

Мастер AppWizard включает файл `Afxolecl.rc` в файл ресурсов программы. В этом файле определены некоторые ресурсы, используемые классами контейнеров OLE.

Он также определяет отдельное меню и соответствующую таблицу акселераторов для каждого из двух режимов, в которых может выполняться программа-контейнер.

Меню и таблица акселераторов `IDR_MAINFRAME` используется, когда нет активного внедренного компонента. Меню содержит обычные команды, которые не относятся к OLE, а также несколько дополнительных команд всплывающего меню `Edit`. К ним относятся команды `Paste Special...`, `Insert New Object...` и `Links...`, а также место для подменю `Object`, помеченное как `<<OLE VERBS GO HERE>>`. В сгенерированном тексте реализованы только команда `Insert New Object...` и команды подменю `Object`.

Меню и таблица акселераторов IDR_CNTR_INPLACE используются для редактирования внедренного компонента "на месте". В этом случае оно объединяется с меню сервера. По некоторым причинам первоначально меню IDR_CNTR_INPLACE отображается редактором в режиме View As Popup. Этот выбор можно отменить, щелкнув на меню правой кнопкой мыши и сняв соответствующий флажок.

Для того, чтобы попробовать программу-контейнер, в работе никаких изменений в код, сгенерированный мастером, можно не вносить.

Тема 21. Создание и применение элементов ActiveX

Элемент ActiveX – это переносимый программный модуль, выполняющий определенную задачу или набор задач. Они похожи на стандартные элементы управления Windows, их можно размещать в диалоговых и других окнах, и они могут реализовывать почти любую функциональность. Для хранения элементов ActiveX используются файлы с расширением .ocx (OCX – прежнее название элементов ActiveX).

Разработанный или полученный откуда-то элемент ActiveX можно включить в программу, разработанную на любом языке программирования, или поместить на Web-страничке, использовать в приложении баз данных Access или любой другой программе, созданной как контейнер ActiveX. При этом программа-контейнер почти так же тесно взаимодействует с элементом, как и с собственным кодом.

Элементы ActiveX предоставляют три основных способа взаимодействия с приложением-контейнером: свойства, методы и события.

- *Свойства* – это атрибуты элемента, которые контейнер может читать или изменять.
- *Методы* – это функции элемента ActiveX, которые может вызывать контейнер.
- *Событие* – это то, что происходит внутри элемента (например, щелчок мышью). Элемент ActiveX обрабатывает это действие, вызывая соответствующую функцию программы-контейнера.

21.1. Разработка элементов ActiveX

В качестве примера мы создадим простой элемент ActiveX (назовем его AXCtrl), предназначенный для отображения рисунка. При щелчке мышью он переключается между двумя версиями изображения. Его свойства будут позволять контейнеру изменять цвет фона элемента, а также добавлять или удалять рамку вокруг рисунка

Генерация файлов программы

Для генерации исходного кода выполним следующие шаги:

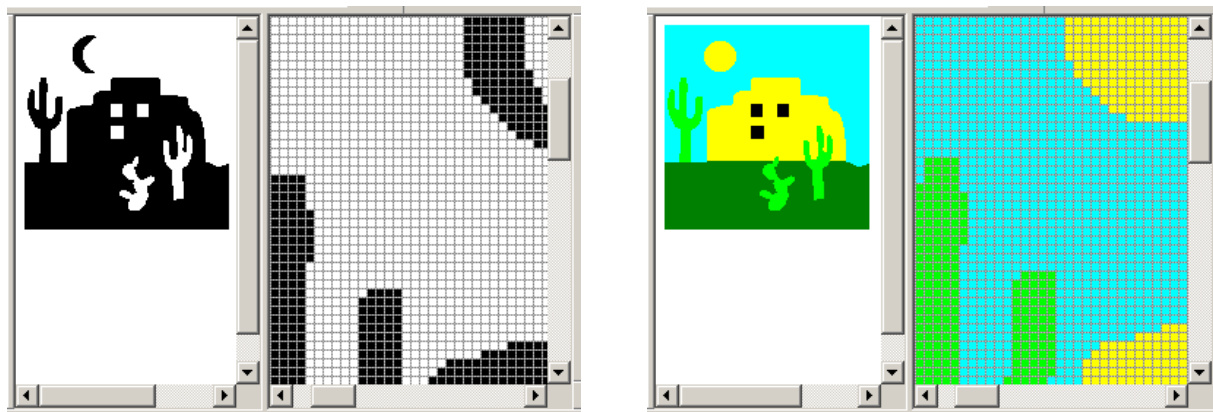
- Выберем команду New... в меню File и откроем вкладку Projects в диалоговом окне New.
- В списке типов проектов выберем MFC ActiveX Control Wizard.
- В поле Project name введем AXCtrl, а в поле Location – путь к папке проекта.
- Убедимся, что выбрана опция Create new workspace и платформа Win32.

- Нажмем кнопку ОК. Мастер ControlWizard отображает последовательно два диалоговых окна для выбора параметров, а затем генерирует файлы с исходным кодом для элемента ActiveX.
- В диалоговом окне Step 1 нажмем кнопку Finish, чтобы согласиться со всеми установками по умолчанию.
- Нажмем кнопку ОК в диалоговом окне New Project Information.

Создание растрового изображения

Два рисунка, отображаемых элементом ActiveX, можно создать с помощью редактора рисунков (команда Resource в меню Insert). Изображение можно нарисовать непосредственно, а можно скопировать через буфер обмена или прочитать из файла. Ниже предлагаются два рисунка размером 140*140 пикселей, которые можно использовать в нашем элементе ActiveX. Они имеют идентификаторы IDB_BITMAP1 и IDB_BITMAP2 соответственно.

При желании можно также настроить изображение, которое появляется на панели элементов при разработке диалогового окна в программном контейнере. Его идентификатор IDB_AXCTRL.



Отображение растровых изображений

Для отображения наших рисунков создаваемым элементом ActiveX следует написать соответствующий код. Сначала добавим три переменные в начало определения класса CAXCtrlCtrl (файл AXCtrlCtl.h):

```
class CAXCtrlCtrl : public COleControl
{
    DECLARE_DYNCREATE(CAXCtrlCtrl)
public:
    CBitmap *m_CurrentBitmap, m_BitmapNight, m_BitmapDay;
    // ...
```

Добавим код загрузки изображений из ресурсов программы в конструкторе класса CAXCtrlCtrl (файл AXCtrlCtl.cpp):

```
CAXCtrlCtrl::CAXCtrlCtrl()
{
    InitializeIIDs(&IID_DAXCtrl, &IID_DAXCtrlEvents);
    // TODO: Инициализируйте данные элемента
    m_BitmapNight.LoadBitmap (IDB_BITMAP1);
    m_BitmapDay.LoadBitmap (IDB_BITMAP2);
    m_CurrentBitmap = &m_BitmapNight; // Отображается первым
}
```

При необходимости перерисовки система вызывает функцию OnDraw класса CAXCtrlCtrl. Сгенерированный мастером код вычерчивает эллипс на белом фоне. Уберем рисование эллипса и добавим свой код (файл AXCtrlCtl.cpp):

```
void CAXCtrlCtrl::OnDraw(CDC* pdc, const CRect& rcBounds,
                        const CRect& rcInvalid)
{
    // TODO: Замените следующий код собственным
    pdc->FillRect(rcBounds,

    CBrush::FromHandle((HBRUSH)GetStockObject(WHITE_BRUSH)));

    BITMAP BM;
    CDC MemDC;
    MemDC.CreateCompatibleDC(NULL);
    MemDC.SelectObject(*m_CurrentBitmap);
    m_CurrentBitmap->GetObject(sizeof(BM), &BM);
    pdc->BitBlt
        ((rcBounds.right - BM.bmWidth) / 2,
         (rcBounds.bottom - BM.bmHeight) / 2,
         BM.bmWidth, BM.bmHeight,
         &MemDC, 0, 0, SRCCOPY);
}
```

Добавление обработчика сообщения о щелчке мыши

В диалоговом окне ClassWizard откроем вкладку Message Maps, в списках Class name и Object IDs выберем класс CAXCtrlCtrl, а в списке Message – идентификатор WM_LBUTTONDOWN и добавим функцию-обработчик (имя по умолчанию – OnLButtonDown). Добавим в нее свой код (файл AXCtrlCtl.cpp):

```

void CAXCtrlCtrl::OnLButtonUp(UINT nFlags, CPoint point)
{
    // TODO: Здесь добавьте собственный код обработчика

    if (m_CurrentBitmap == &m_BitmapNight)
        m_CurrentBitmap = &m_BitmapDay;
    else
        m_CurrentBitmap = &m_BitmapNight;
    InvalidateControl ();

    ColeControl::OnLButtonUp(nFlags, point);
}

```

Наш обработчик изменяет значение переменной `m_CurrentBitmap` так, чтобы она указывала на другое изображение, а затем вызывает функцию `InvalidateControl`, чтобы заставить систему перерисовать изображение (вызовом функции `OnDraw`).

21.2. Определение свойств, методов и событий элементов *ActiveX*

Определение стандартного свойства BackColor

Стандартные свойства (*stock*) входят в набор общих свойств, инициализируемых и сохраняемых MFC. Таковыми являются свойства `Caption`, `Font` и т.п. MFC также отвечает за выполнение необходимых действий при изменении значения свойства.

Для использования стандартного свойства (у нас это `BackColor`) необходимо сделать его доступным, а также написать фрагмент программы, использующий его значение.

Чтобы сделать свойство `BackColor` доступным, воспользуемся мастером `ClassWizard`. Откроем вкладку `Automation`, в списке `Class name` выберем класс элемента `CAXCtrlCtrl` и щелкнем на кнопке `Add Property...` В диалоговом окне `Add Property` из выпадающего списка `External name` выберем стандартное свойство `BackColor`, убедимся, что в области `Implementation` выбрана опция `Stock`, и нажмем кнопку `ОК`, чтобы вернуться на вкладку `Automation`.

Теперь MFC инициализирует это свойство (оно используется для задания цвета окна), хранит его значение и при его изменении форсирует перерисовку соответствующей области, объявляя ее недействительной. При этом использовать цвет, заданный в свойстве, для рисования должен сам программист. Необходимый для этого код мы добавим в функцию `OnDraw`.

Определение свойства ShowFrame

Свойство ShowFrame является *пользовательским*, то есть его нужно создать самостоятельно, присвоить ему имя и написать основную часть кода его поддержки.

Для создания свойства на вкладке Automation диалогового окна ClassWizard в списке Class name выберем класс элемента CAXCtrlCtrl и щелкнем на кнопке Add Property... В появившемся диалоговом окне в поле External name введем ShowFrame, в списке Type выберем значение BOOL и убедимся, что в области Implementation выбрана опция Member variable. Оставим предложенное имя переменной m_showFrame и имя функции OnShowFrameChanged в поле Notification function.

Мастер добавит переменную m_showFrame типа BOOL в определение класса CAXCtrlCtrl и создаст определение и базовую оболочку уведомляющей функции CAXCtrlCtrl::OnShowFrameChanged. Реализующий ее исходный код должен написать программист.

После ввода этих значений щелкнем на кнопке ОК для закрытия диалогового окна Add Property, а затем еще раз, чтобы закрыть окно ClassWizard.

Так как свойство ShowFrame определяется пользователем, его необходимо инициализировать. Сделать это требуется в функции CAXCtrlCtrl::DoPropExchange (файл AXCtrlCtl.cpp):

```
////////////////////////////////////  
// CAXCtrlCtrl::DoPropExchange – реализация устойчивости  
void CAXCtrlCtrl::DoPropExchange(CPropExchange* pPX)  
{  
    ExchangeVersion(pPX, MAKELONG(_wVerMinor, _wVerMajor));  
    COleControl::DoPropExchange(pPX);  
    // TODO: Вызовите функцию PX_ для каждого устойчивого  
    // пользовательского свойства  
    PX_Bool (pPX, _T("ShowFrame"), m_showFrame, FALSE);  
}
```

В уведомляющую функцию OnShowFrameChanged добавим вызов InvalidateControl:

```
void CAXCtrlCtrl::OnShowFrameChanged()  
{  
    // TODO: Здесь добавьте собственный код обработчика  
    InvalidateControl (); // Перерисовать  
    SetModifiedFlag ();   // Элемент изменялся  
}
```

Модификация функции OnDraw

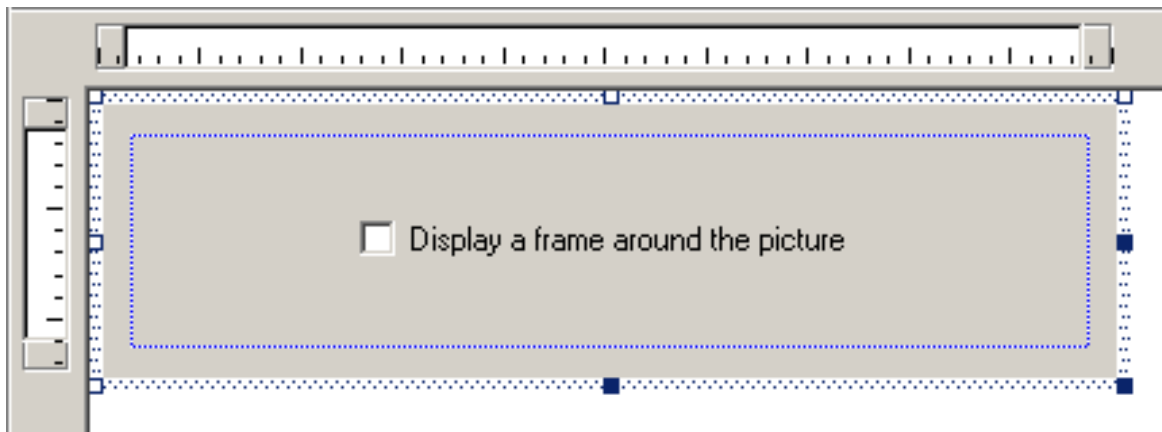
Теперь в файле AXCtrlCtl.cpp необходимо изменить код функции OnDraw так, чтобы она учитывала значения свойств элемента:

```
void CAXCtrlCtl::OnDraw(CDC* pdc, const CRect& rcBounds,
                        const CRect& rcInvalid)
{
    // TODO: Замените следующий код собственным
    CBrush Brush (TranslateColor (GetBackColor ()));
    pdc->FillRect (rcBounds, &Brush); // Рисование фона
    BITMAP BM;
    // Отображение рисунка ...
    if (m_showFrame)
    {
        CBrush *pOldBrush =
            CBrush *)pdc->SelectStockObject (NULL_BRUSH);
        CPen Pen (PS_SOLID | PS_INSIDEFRAME, 10, RGB (0,0,0));
        CPen *pOldPen = pdc->SelectObject (&Pen);
        pdc->Rectangle (rcBounds); // Рисование рамки
        pdc->SelectObject (pOldPen); // Восстановление пера
        pdc->SelectObject (pOldBrush); // Восстановление кисти
    }
}
```

Модификация страницы свойств

Программа элемента ActiveX может содержать одну или несколько *страниц свойств*. Они выглядят как диалоговые окна и содержат набор элементов управления для задания значений свойств элемента. Проектируя программу-контейнер с использованием Visual C++, для задания свойств элемента можно использовать страницы свойств, которые отображаются как вкладки диалогового окна Properties, отображаемого редактором диалоговых окон.

Первоначально проект содержит только одну такую страницу, определенную, как ресурс диалогового окна с идентификатором IDD_PROPPAGE_AXCTRL. Для его редактирования войдем в редактор диалоговых окон, удалим надпись "TODO" и добавим флажок (идентификатор IDC_SHOWFRAME) с надписью, как показано на рисунке.



Теперь для связи флажка со свойством ShowFrame нужно сделать следующее:

- В диалоговом окне мастера ClassWizard откроем вкладку Member Variables.
- В списке Class name выберем имя класса, управляющего страницей свойств CAXCtrlPropPage. В списке Object IDs должен быть идентификатор флажка IDC_SHOWFRAME (он там один).
- Щелкнем по кнопке Add Variable..., чтобы открыть диалоговое окно Add Member Variable.
- В поле Member variable name введем m_ShowFrame. В списке Category выберем значение Value, в списке Variable type значение BOOL. В поле Optional property name введем ShowFrame. Нажмем кнопку OK, чтобы вернуться в диалоговое окно мастера ClassWizard.
- Нажмем кнопку OK, чтобы закрыть диалоговое окно мастера ClassWizard.

Для задания второй страницы свойств, которую можно использовать для выбора стандартного свойства BackColor нужно внести изменения в таблицу свойств элемента

```

////////////////////////////////////
// Страницы свойств
// TODO: Добавьте страницы. Не забудьте увеличить счетчик!
BEGIN_PROPPAGEIDS(CAXCtrlCtrl, 2)
    PROPPAGEID(CAXCtrlPropPage::guid)
    PROPPAGEID(CLSID_CColorPropPage)
END_PROPPAGEIDS(CAXCtrlCtrl)

```

При генерации исходного кода программы мастер AppWizard создал ресурс диалогового окна с идентификатором IDD_ABOUTBOX_AXCTRL и исходный код метода AboutBox, который предназначен для вывода этого диалогового окна программой-контейнером.

В рассматриваемом примере нам не потребуются дополнительные методы (т.е. функции элемента ActiveX, вызываемые программой контейнером), однако в случае необходимости для их создания нужно выполнить следующую последовательность действий:

- В диалоговом окне мастера ClassWizard открыть вкладку Automation.
- В списке Class name выбрать имя класса элемента (в нашем примере это класс CAXCtrlCtrl). Нажать кнопку Add Method...
- Задать имя, тип возвращаемого значения и параметры самого метода, если они есть.
- Нажать кнопку Edit Code (или иным образом перейти в режим редактирования) и ввести собственно исходный код метода.

Определение событий

Определение события состоит в назначении функции, которую должна выполнить программа-контейнер в ответ на то, что данное событие произошло. Этот вызов принято называть *активизацией события*. Упомянутые функции называются функциями Fire... (например, FireClick).

Так же, как и свойство или метод, событие может быть пользовательским или стандартным. Для стандартных событий реализация соответствующей функции Fire... содержится в библиотеке MFC. В качестве примера рассмотрим добавление функции FireClick, реализация которой предоставляется библиотекой MFC (как член класса COleControl).

- В диалоговом окне мастера ClassWizard откроем вкладку ActiveX Events.
- Нажмем кнопку Add Event...
- В списке External name выберем стандартное событие Click. При этом в поле Internal name отобразится имя FireClick. Это имя используется программой элемента ActiveX для активизации события, изменить его нельзя.
- Убедимся, что в области Implementation отмечена опция Stock (стандартное событие). Щелкнем на кнопке ОК в диалоговом окне Add Event, а затем в окне мастера ClassWizard.

Теперь можно построить элемент ActiveX, щелкнув на кнопке Build. Результатом станет построение файла AXCtrl.osx, который автоматически регистрируется в системе, что сделает возможным доступ к нему со стороны программы-контейнера. Для установки элемента ActiveX, полученного другим путем, достаточно скопировать его (файл .osx, а также .hlp и .lic, если есть) в системный каталог, и вызвать системную утилиту Regsvr32, указав в командной строке имя osx-файла.

21.3. Разработка программы-контейнера элементов ActiveX

В качестве примера мы создадим программу-контейнер AXCont для разработанного ранее элемента ActiveX. Программа будет построена на основе диалогового окна, в котором отображается элемент ActiveX. Программа считывает и устанавливает свойства BackColor и ShowFrame, вызывает метод AboutBox и подает звуковой сигнал в ответ на событие Click.

Генерация файлов с исходным текстом

Для генерации исходного кода выполним следующие шаги:

- Выберем команду New... в меню File и откроем вкладку Projects в диалоговом окне New.
- В списке типов проектов выберем MFC AppWizard (exe). В поле Project name введем AXCont, а в поле Location – путь к папке проекта. Убедимся, что выбрана опция Create new workspace и платформа Win32. Нажмем кнопку ОК.
- В диалоговом окне мастера Step 1 выберем установку Dialog based.
- В диалоговом окне мастера Step 2 отменим выбор опции About Box, но оставим выбранной опцию ActiveX Controls. В текстовое поле "Please enter a title for your dialog" введем заголовок ActiveX Control Container Demo.
- В следующем диалоговом окне (Step 3) выберем статическую компоновку библиотеки MFC
- В диалоговом окне Step 4 оставим все без изменений.

Добавление элемента ActiveX в проект

Прежде чем отобразить в программе-контейнере элемент ActiveX, его необходимо добавить в проект. Добавление элемента генерирует "класс-упаковку" (wrapper class) – в нашем случае это класс SAXCtrl, позволяющий программе взаимодействовать с элементом, а также добавляет в панель инструментов Controls редактора диалоговых окно кнопку, позволяющую добавлять элемент в диалоговое окно.

Для добавления нашего элемента ActiveX сделаем следующие действия:

- В меню Project войдем в подменю Add To Project и выберем команду Components and Controls..., чтобы открыть диалоговое окно Components and Controls Gallery.
- В появившемся диалоговом окне убедимся, что выбрана папка Gallery и откроем в ней папку Registered ActiveX Controls.
- Выберем имя нужного элемента ActiveX (AXCtrl Control в нашем случае).
- Щелкнем на кнопке Insert, а затем подтвердим свой выбор, нажав ОК.

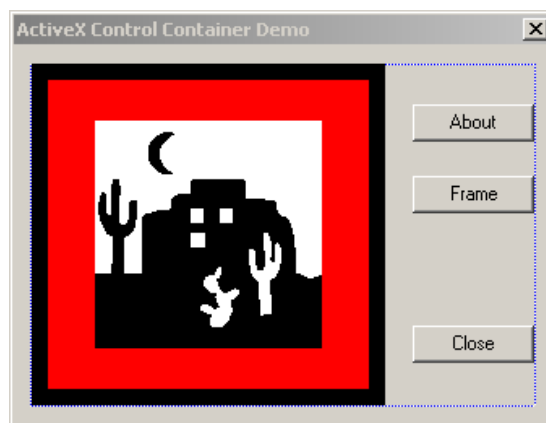
- В диалоговом окне Confirm Classes появятся имя класса-упаковки и имена файлов, в которых класс определен. Их можно изменить, однако мы согласимся с предложенными именами, нажав OK.
- Щелкнем на кнопке Close в диалоговом окне Components and Controls Gallery.

Проектирование диалогового окна программы

Если после создания исходных файлов программы мастер AppWizard не открыл для редактирования главное диалоговое окно программы (идентификатор IDD_AXCONT_DIALOG), сделаем это самостоятельно.

Для проектирования главного окна программы выполним следующие действия:

- Удалим надпись "TODO" и кнопку OK.
- Заменим свойство Caption кнопки Cancel на Close и добавим две кнопки: одну с надписью About и идентификатором IDC_ABOUT, и другую – с надписью Frame и идентификатором IDC_FRAME.
- Добавим элемент ActiveX в диалоговое окно, щелкнув сначала на кнопке OCX в нижней части панели инструментов Controls, а затем в нужном месте диалогового окна. Окончательный вид окна показан на рисунке.
- Изменим свойства элемента ActiveX, щелкнув на нем правой кнопкой и выбрав команду Properties. В нашем случае диалоговое окно AXCtrl Control Properties состоит из двух страниц свойств, добавленных или измененных при создании элемента, а именно, страницы свойств Color и заданной по умолчанию страницы свойств.
- Откроем страницу свойств Color и для примера изменим свойство BackColor на какое-либо другое значение, например, назначим цветом фона красный цвет. Для этого достаточно щелкнуть на ярлычке Color и выбрать нужный цвет. Таким образом осуществляется изменение начальных установок стандартного свойства элемента.
- Изменим начальную установку пользовательского свойства ShowFrame. Для этого откроем страницу свойств программы, щелкнув на вкладке Control, а затем установим флажок выбора опции Display a frame around the picture. Эти действия приводят к начальной установке значения свойства ShowFrame в TRUE.
- Стандартные значения остальных свойств оставим неизменными.



Присоединение компонента к объекту класса-упаковки

Теперь необходимо в диалоговом окне присоединить элемент ActiveX к экземпляру класса-упаковки. Это позволяет использовать функции класса-упаковки для изменения свойств элемента и для вызова его методов.

Выполним следующие действия:

- В диалоговом окне мастера ClassWizard откроем вкладку Member Variables.
- В списке Class name выберем класс диалогового окна CAXContDlg.
- В списке Control IDs выберем идентификатор компонента ActiveX IDC_AXCTRLCTRL1.
- Щелкнем по кнопке Add Variable..., чтобы открыть диалоговое окно Add Member Variable.
- В поле Member variable name введем m_AXCtrl – имя объекта класса-упаковки. В списке Category выберем значение Control, в списке Variable type – имя класса-упаковки CAXCtrl (это единственный возможный вариант).
- Нажмем кнопку ОК, чтобы вернуться в диалоговое окно мастера ClassWizard. Оставим его пока открытым для определения обработчиков сообщений для кнопок.

Определение обработчиков сообщений для кнопок

Теперь необходимо определить обработчики сообщений для кнопок About и Frame, отображаемых в диалоговом окне программы. Выполним следующие действия:

- Откроем вкладку Message Maps в диалоговом окне мастера ClassWizard. В списке Class name выберем класс диалогового окна CAXContDlg.
- В списке Control IDs выберем идентификатор кнопки IDC_FRAME, в списке Messages значение BN_CLICKED. Щелкнем на кнопке Add Function... и согласимся с предложенным именем OnFrame.
- Аналогичным образом добавим обработчик OnAbout для кнопки About (идентификатор IDC_ABOUT).

Отредактируем код добавленных обработчиков следующим образом:

```
void CAXContDlg::OnFrame()  
{  
    // TODO: Здесь добавьте собственный код обработчика  
    m_AXCtrl.SetShowFrame (!m_AXCtrl.GetShowFrame ());  
}
```

```
void CAXContDlg::OnAbout ()
{
    // TODO: Здесь добавьте собственный код обработчика
    m_AXCtrl.AboutBox ();
}
```

Вызываемые в коде первого обработчика функции **GetShowFrame** и **SetShowFrame** сгенерированы мастером при создании класса оболочки **CAXCtrl** (файлы **AXCtrl.h** и **AXCtrl.cpp**). Рекомендуется ознакомиться с текстом этих и других функций данного класса, чтобы понять, с помощью каких вызовов происходит взаимодействие контейнера со свойствами и методами компонента **ActiveX**.

Определение обработчика события *Click*

Это наше последнее действие при создании демонстрационной программы-контейнера. Выполним следующие действия:

- Откроем вкладку **Message Maps** в диалоговом окне мастера **ClassWizard**. В списке **Class name** выберем класс диалогового окна **CAXContDlg**.
- В списке **Control IDs** выберем идентификатор **IDC_AXCTRLCTRL1** элемента **ActiveX**.
- В списке **Messages** имя **Click** (единственное в этом списке, так как мы не добавляли других событий). Щелкнем на кнопке **Add Function...** и согласимся с предложенным именем **OnClickAxctrlctrl1**.

Отредактируем код добавленного обработчика следующим образом:

```
void CAXContDlg::OnClickAxctrlctrl1 ()
{
    // TODO: Здесь добавьте собственный код обработчика
    ::MessageBeep (MB_OK); // Звуковой сигнал
}
```

Тема 22. Динамически подключаемые библиотеки

22.1. Основы DLL. Экспорт и импорт функций

Если C++ классы обеспечивают модульность процесса разработки программы, то *динамически подключаемые библиотеки* (Dynamic-Link Library, DLL) обеспечивают модульность программы *в период ее выполнения*. В самой Windows поддержка основных функций строится на применении именно DLL.

DLL-модуль – это файл на диске (обычно с расширением .dll), который состоит из глобальных данных, откомпилированных функций и ресурсов и который становится частью вашего процесса. В DLL содержатся так называемые *экспортируемые* (export) функции, которые *импортирует* (import) клиентская программа. DLL-модули в Win32 позволяют экспортировать еще и глобальные переменные.

В Win32 каждый процесс получает свою копию глобальных переменных DLL для чтения и записи. Если требуется, чтобы несколько процессов совместно использовали участок памяти, то следует прибегнуть к одному из способов, рассмотренных в теме "Связи между процессами". Если DLL запрашивает память из кучи, то эта память выделяется из кучи, принадлежащей клиентскому процессу.

Согласование импортируемых элементов с экспортируемыми

DLL содержит таблицу экспортируемых функций. Они идентифицируются извне по их символьному имени и (не обязательно) *порядковому номеру*. В таблице хранятся адреса этих функций в пределах DLL. Клиентская программа знает символьные имена функций либо их порядковые номера. Процесс динамической компоновки состоит в формировании таблицы, связывающей вызовы клиентской программы с адресами функций в DLL.

В коде DLL экспортируемые функции следует объявить явным образом, например:

```
extern "C" __declspec(dllexport) int MyFunction (int n);
```

Вместо этого можно перечислить экспортируемые функции в файле определения модуля (.def), однако это более хлопотно. В клиентской программе нужно указывать, какая функция импортируется, например:

```
extern "C" __declspec(dllimport) int MyFunction (int n);
```

Модификатор extern "C" требует от компилятора использования простого имени (а не *расширенного*, которое нельзя использовать в других языках). По умолчанию компилятор формирует передачу параметров по

правилам языка C. Если требуется придерживаться правил Паскаля, то понадобится модификатор `__stdcall`.

Кроме того, в проекте клиентской программы следует определить *библиотеку импорта* (.lib) для компоновщика, а сама программа должна содержать как минимум один явный вызов функции, импортируемой из DLL.

Явное и неявное связывание

Рассмотренные примеры описаний относились к *неявному связыванию* (implicit linking), по-видимому, применяющемуся чаще. При создании DLL-файла компоновщик создает дополнительный LIB-файл, содержащий символьные имена всех экспортируемых элементов и (возможно) их порядковые номера. Он также содержит имя DLL-файла без указания пути. При загрузке клиента Windows находит и загружает нужные DLL-модули, а затем динамически связывает их по символьным именам или порядковым номерам.

При *явном связывании* (explicit linking) файл импорта не используется – вместо этого вызывается Win32-функция `::LoadLibrary` с полным именем DLL-файла в качестве параметра. Затем вызывается `::GetProcAddress` для получения адреса импортируемой функции. Допустим, в DLL-файле функция описана так:

```
extern "C" __declspec(dllexport) double SquareRoot(double n);
```

Тогда явное связывание может выглядеть, например, так:

```
typedef double (SQRTPROC)(double);
HINSTANCE hInstance;
SQRTPROC* pFunction;
VERIFY(hInstance=::LoadLibrary("C:\\MyProgs\\MyDll.dll"));
VERIFY(pFunction=(SQRTPROC*)::GetProcAddress
        ((HMODULE)hInstance, "SquareRoot"));
double d=(*pFunction)(81.0); // Вызов функции SquareRoot
```

Точка входа в DLL: функция DllMain

По умолчанию компоновщик считает основной точкой входа в DLL функцию `_DllMainCRTStartup`. Windows, загружая DLL, вызывает эту функцию, а та сначала вызывает конструкторы глобальных объектов, а затем глобальную функцию `DllMain`. Последняя вызывается не только при подключении к процессу, но и при отключении от него. Ее заготовка (если требуется) может выглядеть примерно так:


```

HINSTANCE g_hInstance;
extern "C" int APIENTRY DllMain
    (HINSTANCE hInstance, DWORD dwReason, LPVOID lpReserved)
{
    if (dwReason == DLL_PROCESS_ATTACH)
    {
        TRACE0("EXAMPLE.DLL Initializing!\n");
        g_hInstance = hInstance;
        // Здесь реализуйте инициализацию
    }
    else if (dwReason == DLL_PROCESS_DETACH)
    {
        TRACE0("EXAMPLE.DLL Terminating!\n");
        // Здесь реализуйте очистку
    }
    return 1;    // OK
}

```

При отсутствии в программе функции DllMain вместо нее подставляется заглушка.

Порядок поиска DLL клиентской программой

Если в программе используется явная компоновка с использованием LoadLibrary, то при вызове этой функции можно указать полное имя DLL-модуля. Если полный путь не указан, а также в случае неявной компоновки Windows ищет DLL в следующих местах (по порядку):

- В каталоге, содержащем данный EXE-файл.
- В текущем каталоге процесса.
- В системном каталоге Windows.
- В каталоге Windows.
- В каталогах, прописанных в переменной окружения PATH.

Замечание. Поскольку обычно DLL и клиентская программа создаются в рамках разных проектов, при внесении изменений в DLL не забывайте скопировать ее новую версию в то место, откуда ее берет клиент.

Отладка DLL

Достаточно просто запустить отладчик в проекте DLL. При первом запуске отладчик запросит путь к клиентской программе. После этого он его запомнит и будет запускать программу автоматически. Однако при этом, если в клиенте явно не указан путь к DLL, нужно или копировать DLL-модуль, или соответствующим образом настроить переменную окружения PATH.

22.2. Пример создания и использования DLL

Создание модуля DLL

В качестве примера создадим DLL, содержащую единственную экспортируемую функцию – вычисление квадратного корня. Для генерации исходного кода воспользуемся, как обычно, мастером AppWizard.

При выборе типа проекта укажем MFC AppWizard (dll), в качестве имени возьмем SRdll. На этапе Step 1 выберем вариант Regular DLL using shared MFC DLL и щелкнем на кнопке Finish.

В сгенерированном файле SRdll.cpp добавим в начале строку

```
#include <math.h>
```

В конце файла поместим экспортируемую функцию ExampleSquareRoot:

```
extern "C" __declspec(dllexport)
double ExampleSquareRoot(double d)
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());
    TRACE("Entering ExampleSquareRoot\n");
    if (d >= 0.0)
        return sqrt(d);
    AfxMessageBox
        ("Can't take square root of a negative number.");
    return 0.0;
}
```

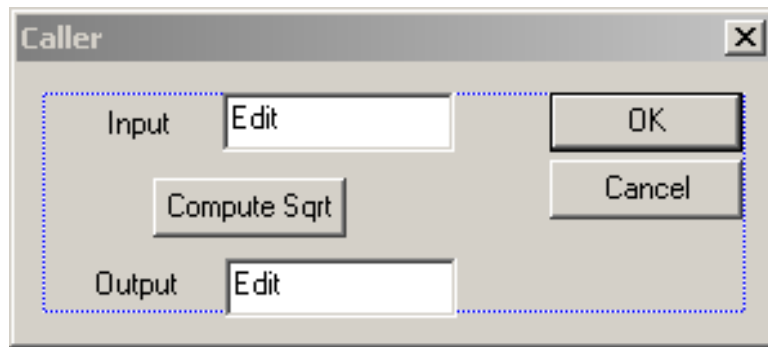
Создание программы, использующей DLL

Программа-клиент будет организована как диалоговое приложение. Для генерации исходного кода вызовем диалоговое окно мастера AppWizard.

При выборе типа проекта укажем MFC AppWizard (exe), в качестве имени возьмем Caller. На этапе Step 1 выберем вариант Dialog based. На этапе Step 2 снимем флажки с опций About и ActiveX Controls, затем щелкнем на кнопке Finish.

В окне диалогового редактора уберем надпись TODO из предложенной формы диалога IDD_CALLER_DIALOG.

После этого добавим новые элементы: два поля ввода (идентификаторы IDC_INPUT и IDC_OUTPUT) с надписями "Input" и "Output" соответственно и кнопку с надписью "Compute Sqrt" (идентификатор IDC_COMPUTE). Окончательный вид диалогового окна показан на рисунке.



В файле CallerDlg.h перед объявлением класса диалога CCallerDlg поместим объявление импортируемой функции:

```
extern "C" __declspec(dllimport)
double ExampleSquareRoot(double d);
```

Следующий шаг – добавление переменных для работы с полями ввода и обработчика нажатия кнопки Compute Sqrt.

Войдем в диалоговое окно мастера ClassWizard и выберем вкладку Member Variables. Для полей ввода IDC_INPUT и IDC_OUTPUT добавим переменные (категория Value, тип double) с именами m_dInput и m_dOutput соответственно.

Перейдем на вкладку Message Maps и для кнопки IDC_COMPUTE добавим функцию-обработчик сообщения BN_CLICKED, имя по умолчанию – OnCompute. Введем код функции следующим образом (файл CallerDlg.cpp):

```
void CCallerDlg::OnCompute()
{
    // TODO: Добавьте собственный код обработчика
    UpdateData(TRUE); // Получить данные из диалогового окна
    m_dOutput = ExampleSquareRoot(m_dInput);
    UpdateData(FALSE); // Обновить данные в диалоговом окне
}
```

Последнее, что нужно сделать, – это добавить библиотеку импорта в список входных библиотек компоновщика. Проще всего для этой цели вызвать в меню Project команду Add To Project и в появившемся диалоговом окне выбрать файл SRdll.lib.

22.3. DLL-расширения и обычные DLL

В рассмотренном примере мы использовали *обычную DLL* (regular DLL). Между тем AppWizard позволяет создавать еще и *DLL-расширения* (extension DLL).

Основное различие их состоит в том, что DLL-расширения поддерживают интерфейс C++ и позволяют экспортировать целые классы, так что

клиент может создавать экземпляры этих классов или разрабатывать производные от них классы.

DLL-расширение динамически связывается с MFC, поэтому необходимо, чтобы клиентская программа компоновалась с MFC тоже динамически, причем версии динамических библиотек у нее и DLL-расширения должны совпадать. Если же требуется DLL, которую можно было бы использовать в любой Win32-среде программирования, то следует использовать обычную DLL, позволяющую экспортировать только C-функции. Разумеется, в самой DLL можно использовать классы C++ и классы MFC в том числе.

Если DLL-расширение должно содержать только экспортируемые классы, то создать и использовать его довольно легко. С помощью AppWizard создается заготовка кода, содержащая только функцию `DllMain`. Добавляя в проект свои классы, их следует объявлять с использованием макроса `AFX_EXT_CLASS`, например:

```
class AFX_EXT_CLASS CMyClass : public CObject
```

Это изменение нужно внести в заголовочный файл (.h), используемый как для DLL, так и для программы-клиента. Макрос генерирует код в зависимости от ситуации: либо класс экспортируется из DLL, либо импортируется клиентом.

Дополнительную информацию по данному вопросу можно найти в документации и специальной литературе

Литература

1. Янг Майкл Дж. Visual C++ 6. Полное руководство: в 2 т. - Киев: Издательская группа BHV, 1999.
2. Круглински Д., Уингоу С., Шеферд Дж. Программирование на Microsoft Visual C++ 6.0 для профессионалов. - СПб: Питер, 2001.
3. Олафсен Юджин, Скрайбер Кенн, Уайт К.Дэвид и др. MFC и Visual C++ 6. Энциклопедия программиста. - СПб: ООО "ДиаСофтЮП", 2004.
4. Рихтер Дж. Windows для профессионалов: создание эффективных Win32-приложений с учетом специфики 64-разрядной версии Windows. - СПб: Питер, 2001.

Оглавление

Введение	3
Тема 1. Установка программного обеспечения	4
1.1. Установка Microsoft Visual C++ 6	4
1.2. Установка справочной системы Visual C++ 6	6
Тема 2. Создание программ в среде Developer Studio	7
Создание проекта	7
Создание и редактирование исходного файла программы	7
Некоторые возможности текстового редактора	7
Отладка программы	8
Комбинации клавиш отладчика Developer Studio	9
Тема 3. Модель программирования в Windows	10
Обработка сообщений	10
Интерфейс графического устройства	10
Программирование, основанное на ресурсах	10
Динамически подключаемые библиотеки	11
Интерфейс прикладных программ Win32 API	11
Тема 4. Процесс построения программ в Visual C++	12
4.1. Создание программы в Visual C++	12
Проект программы	12
Промежуточные файлы Visual C++	13
4.2. Компоненты Visual C++	13
Редакторы и средства просмотра ресурсов	13
Компилятор C/C++	14
Редактор исходного текста	14
Компилятор ресурсов	14
Компоновщик	14
AppWizard	15
ClassWizard	15
Средства просмотра исходного кода	15
Интерактивная справочная система	15
Components and Controls Gallery	15

Тема 5. Создание программ с графическим интерфейсом	16
Генерация исходного кода.....	16
Изменение исходного кода.....	17
Классы и файлы программы.....	19
Этапы выполнения программы	19
Тема 6. Реализация представления	22
<i>6.1. Реализация графического представления.....</i>	<i>22</i>
Генерация исходных файлов	22
Определение переменных класса представления	22
Инициализация переменных класса представления	22
Идентификаторы стандартных указателей Windows, которые можно передавать функции LoadStandardCursor.....	23
Добавление обработчиков сообщений Windows.....	23
Командные сообщения.....	23
Пример: добавление обработчика нажатия левой кнопки мыши.....	24
Программирование обработки нажатия левой кнопки мыши	25
Схема сообщений	25
Схема сообщений – функция OnMouseMove	26
Схема сообщений – функция OnLButtonUp	26
Параметры сообщений мыши	27
Проектирование ресурсов программы	27
Настройка окна программы	27
<i>6.2. Реализация текстового представления</i>	<i>28</i>
Генерация исходных файлов	29
Редактирование ресурсов программы	29
Редактирование таблицы горячих клавиш.....	29
Тема 7. Реализация документа	30
<i>7.1. Сохранение графических данных</i>	<i>30</i>
Определение класса для сохранения информации о введенных линиях.....	30
Дополнения в классе документа	30
Реализация функций класса документа.....	31
<i>7.2. Перерисовка окна</i>	<i>32</i>
<i>7.3. Добавление команд в меню</i>	<i>32</i>
<i>7.4. Удаление данных документа.....</i>	<i>33</i>
<i>7.5. Реализация команд меню</i>	<i>33</i>
Обработка команды Delete All	33
Обработка команды Undo	34

Тема 8. Хранение данных	36
8.1. Ввод-вывод программы <i>MiniDraw</i>	36
Добавление команд в меню File	36
Задание стандартного расширения файлов	36
Поддержка команд меню File	37
Сериализация данных документа	38
Установка флага изменений	39
Поддержка технологии "drag-and-drop"	40
Регистрация типа файла	40
8.2. Ввод-вывод программы <i>MiniEdit</i>	41
Добавление команд в меню File	41
Добавление кода поддержки	41
Функции Read и Write класса CArchive	42
8.3. Другие средства ввода-вывода файлов	42
Тема 9. Прокрутка и разделение окон представления	43
9.1. Добавление средств прокрутки окна	43
Преобразование координат	44
Ограничение размера рисунка	45
Изменение формы указателя	47
9.2. Добавление средств разделения окна	48
9.3. Обновление окна представления	49
Эффективная перерисовка	50
Тема 10. Перемещаемые панели и строки состояния	53
10.1. Добавление в новую программу перемещаемой панели инструментов и строки состояния	53
10.2. Добавление перемещаемой панели инструментов в программу <i>MiniDraw</i>	54
Определение ресурсов	55
Добавление новых команд меню	55
Изменение текста программы	57
Написание обработчиков сообщений	59
Реализация обработчиков сообщений	59
10.3. Добавление строки состояния в программу <i>MiniDraw</i>	61
Необходимые объявления	61
Завершение создания меню View	62
Изменение интерактивной справки	62

Тема 11. Создание диалоговых окон	63
<i>11.1. Создание модальных диалоговых окон</i>	<i>63</i>
Создание программы	64
Диалоговое окно Format и его элементы управления	66
Задание порядка обхода элементов управления	66
Создание класса для управления диалоговым окном	67
Определение переменных-членов класса	67
Определение обработчиков событий	68
Управление диалоговым окном класса CFormat	69
MFC-классы для элементов управления	70
Управление окном класса CFormat – Style	70
Управление окном класса CFormat – Justify и Pitch	71
Управление окном класса CFormat – OnChangeSpacing	71
Управление окном класса CFormat – OnPaint	72
Отображение диалогового окна	73
Отображение диалогового окна – OnTextFormat	74
Отображение окна класса CFontDemoView – OnDraw	75
<i>11.2. Создание немодальных диалоговых окон</i>	<i>77</i>
<i>11.3. Создание диалоговых окон с вкладками</i>	<i>78</i>
Создание шаблона диалогового окна	78
<i>11.4. Диалоговые окна общего назначения</i>	<i>82</i>
Тема 12. Разработка диалоговых приложений	83
<i>12.1. Простые диалоговые программы</i>	<i>83</i>
Генерация исходных файлов программы DlgDemo	83
Настройка программы DlgDemo	84
Функция InitInstance	87
<i>12.2. Программы просмотра форм</i>	<i>87</i>
Генерация исходных файлов	88
Настройка программы FormDemo	88
Тема 13. Создание многодокументных приложений	92
<i>13.1. Многодокументный интерфейс</i>	<i>92</i>
<i>13.2. Создание MDI-программы в среде Developer Studio</i>	<i>92</i>
Генерация кода	92
<i>13.3. Основные классы MDI-программы</i>	<i>93</i>
Класс приложения	93
Класс документа	93
Класс главного окна	93
Класс дочернего окна	94

Использование документов различных типов.....	95
Класс представления	95
Сгенерированный код программы.....	95
<i>13.4. Настройка ресурсов.....</i>	<i>96</i>
Команда New Window	96
Добавление горячих клавиш и значка	97
Тема 14. Ввод/вывод символов	98
<i>14.1. Отображение текста.....</i>	<i>98</i>
Генерация исходного кода программы	98
Отображение текста в окне представления	98
Основные этапы отображения текста внутри окна представления.....	100
Метрики шрифта.....	100
Цвет шрифта.....	100
Функции класса CDC для установки и определения атрибутов текста.....	101
Функции отображения текста	101
Создание объекта Font и сохранение текста.....	102
Использование стандартных шрифтов.....	103
Значения nIndex для выбора стандартных шрифтов	104
Поддержка средств прокрутки	104
<i>14.2. Чтение кодов символов, вводимых с клавиатуры</i>	<i>106</i>
Обработка сообщения WM_KEYDOWN.....	106
О работе функции CTextDemoView::OnKeyDown.....	108
Обработка сообщения WM_CHAR.....	109
<i>14.3. Управление курсором при редактировании.....</i>	<i>112</i>
Добавление новых функций обработки сообщений	112
Добавление кода функций-обработчиков	112
Тема 15. Использование функций рисования	115
<i>15.1. Создание объекта контекста устройства</i>	<i>115</i>
<i>15.2. Выбор средств рисования внутри объекта</i>	<i>116</i>
Перо и кисть	116
Выбор стандартных инструментов рисования	117
Создание инструментов рисования	118
Создание кисти	119
Выбор пера или кисти в объекте контекста устройства.....	120
Пример функции OnDraw	120
<i>15.3. Установка атрибутов рисования для объекта.....</i>	<i>121</i>
Стандартные атрибуты.....	121
Режим отображения.....	121

<i>15.4. Создание графических изображений</i>	123
Базовые функции рисования	123
Программа Mandel – постановка задачи	123
Программа Mandel – генерация и настройка кода	124
<i>15.5. Функции рисования - члены класса CDC</i>	127
Прямые линии	127
Регулярные кривые – дуга	128
Регулярные кривые – кривая Безье	128
Режим рисования линий, режим фона	130
Рисование замкнутых линий	131
Другие функции рисования	132
<i>15.6. Пример – программа MiniDraw</i>	133
Изменения в интерфейсе	133
Определение классов для фигур	134
Определение классов для фигур – комментарии	135
Другие модификации программы	136
Тема 16. Растровые изображения и битовые операции	137
<i>16.1. Создание растровых изображений</i>	137
Загрузка растрового изображения из ресурсов	137
Создание растрового изображения с использованием функций рисования	138
Пример создания растрового изображения с использованием функций рисования	139
Отображение растрового изображения	140
Другие способы использования растровых изображений	141
<i>16.2. Выполнение битовых операций при отображении</i>	141
Функция PatBlt	142
Функция BitBlt	142
Использование функции BitBlt для анимации	144
Функция StretchBlt	145
<i>16.3. Отображение значков</i>	146
Пример – программа BitDemo	147
Тема 17. Печать и предварительный просмотр	150
<i>17.1. Добавление в программу средств печати и предварительного просмотра</i>	150
Добавление средств печати при генерации кода	150
Модификация ресурсов программы	150
Модификация текста программы	151
Добавление средств печати в окно представления класса CEditView ..	152

17.2. Усовершенствованная печать	153
Изменение размера рисунка	153
Схема процесса печати и предварительного просмотра	154
Виртуальные функции печати и их назначение	154
Переопределение виртуальных функций печати	158
Переопределение виртуальной функции OnBeginPrinting.....	158
Переопределение виртуальной функции OnPrepareDC	159
Модификация виртуальной функции OnDraw	160
Функция GetDeviceCaps.....	160
Тема 18. Многопоточные приложения	162
18.1. Создание и управление вторичными потоками	162
Настройка среды разработчика	162
Создание и запуск вторичного потока	163
Прекращение выполнения потока	164
Управление потоком	164
18.2. Особенности использования MFC-классов в многопоточных программах.....	165
Ограничения на использование MFC-классов	165
Создание потоков пользовательского интерфейса	166
18.3. Синхронизация потоков.....	167
Использование мьютексов.....	167
Другие объекты синхронизации	168
Другие типы синхронизации	169
Многопоточная программа MandelMT	170
Тема 19. Связи между процессами	176
19.1. Запуск новых процессов	176
Функция ::CreateProcess.....	176
Использование дескриптора процесса	177
19.2. Синхронизация процессов	178
Получение дескриптора процесса.....	179
Наследуемые дескрипторы.....	179
Дублируемые дескрипторы	180
19.3. Обмен данными между процессами.....	180
Обмен данными по каналам	180
Совместное использование памяти	181
19.4. Использование буфера обмена для передачи данных	182
Команды буфера обмена.....	182
Использование буфера обмена для переноса текста.....	183
Получение текста из буфера обмена	186

Копирование растрового изображения в буфер обмена.....	188
Получение растрового изображения из буфера обмена.....	189
Использование буфера обмена для передачи данных зарегистрированных форматов	191
Тема 20. Механизм OLE.....	192
<i>20.1. Внедрение, связывание и автоматизация</i>	<i>192</i>
<i>20.2. Разработка программы-сервера</i>	<i>194</i>
Генерация исходного текста программы	194
Класс приложения	194
Класс документа	196
Класс компонента сервера.....	197
Класс окна редактирования на месте	197
Класс представления	198
Ресурсы	198
Добавление кода, реализующего основные функции.....	198
Добавление поддержки OLE – меню.....	199
Добавление поддержки OLE – исходный код	200
<i>20.3. Разработка программы-контейнера</i>	<i>202</i>
Класс приложения	202
Класс документа	203
Класс компонента контейнера	203
Класс представления	204
Ресурсы	205
Тема 21. Создание и применение элементов ActiveX.....	207
<i>21.1. Разработка элементов ActiveX</i>	<i>207</i>
Генерация файлов программы	207
Создание растрового изображения.....	208
Отображение растровых изображений.....	208
Добавление обработчика сообщения о щелчке мыши	209
<i>21.2. Определение свойств, методов и событий элементов ActiveX.....</i>	<i>210</i>
Определение стандартного свойства BackColor	210
Определение свойства ShowFrame	211
Модификация функции OnDraw	212
Модификация страницы свойств	212
Определение событий	214
<i>21.3. Разработка программы-контейнера элементов ActiveX</i>	<i>215</i>
Генерация файлов с исходным текстом	215
Добавление элемента ActiveX в проект	215
Проектирование диалогового окна программы	216
Присоединение компонента к объекту класса-упаковки	217

Определение обработчиков сообщений для кнопок.....	217
Определение обработчика события Click	218
Тема 22. Динамически подключаемые библиотеки.....	219
<i>22.1. Основы DLL. Экспорт и импорт функций</i>	<i>219</i>
Согласование импортируемых элементов с экспортируемыми	219
Явное и неявное связывание	220
Точка входа в DLL: функция DllMain	220
Порядок поиска DLL клиентской программой	221
Отладка DLL	221
<i>22.2. Пример создания и использования DLL.....</i>	<i>222</i>
Создание модуля DLL	222
Создание программы, использующей DLL	222
<i>22.3. DLL-расширения и обычные DLL</i>	<i>223</i>
Литература.....	225

Учебное издание

Васильчиков Владимир Васильевич

Программирование в Visual C++ с использованием библиотеки MFC

Учебное пособие

Редактор, корректор А.А. Аладьева

Подписано в печать 20.06.2006 г. Формат 60х84/16.
Бумага тип. Усл. печ. л. 13,95. Уч.-изд. л. 8,96.
Тираж 100 экз. Заказ

Оригинал-макет подготовлен
в редакционно-издательском отделе ЯрГУ.

Ярославский государственный университет.
150000 Ярославль, ул. Советская, 14.

Отпечатано
ООО «Ремдер» ЛР ИД № 06151 от 26.10.2001.
г. Ярославль, пр. Октября, 94, оф. 37
тел. (4852) 73-35-03, 58-03-48, факс 58-03-49.

