Учебно-методические пособия Учебно-научного центра ОИЯИ Дубна

УНЦ-2010-45

В. А. Калинников

# РАЗРАБОТКА WINDOWS-ПРИЛОЖЕНИЙ С ПОМОЩЬЮ MFC-БИБЛИОТЕКИ КЛАССОВ В СРЕДЕ ПРОГРАММИРОВАНИЯ MICROSOFT VISUAL C++2008

2010

Учебно-научный центр ОИЯИ



В. А. Калинников



# РАЗРАБОТКА WINDOWS-ПРИЛОЖЕНИЙ С ПОМОЩЬЮ MFC-БИБЛИОТЕКИ КЛАССОВ В СРЕДЕ ПРОГРАММИРОВАНИЯ MICROSOFT VISUAL C++2008

*Учебное пособие*



Дубна 2010

**Калинников В. А.**

К17 Разработка Windows-приложений с помощью MFC-библиотеки классов в среде программирования Microsoft Visual C++2008: Учебное пособие. — Дубна: ОИЯИ, 2010. — 136 с.

ISBN 978-5-9530-0262-2

В пособии рассмотрена технология разработки Windows-приложений с использованием MFC-библиотеки классов в среде программирования Microsoft Visual Studio C++2008. Подробно описана структура и технология создания разрабатываемого приложения и его основных компонентов, таких как дочерние и диалоговые окна, меню, панели инструментов, управляющие элементы, подключаемые библиотеки и редактор ресурсов. Приводятся примеры программ с подробными комментариями.

**Kalinnikov V. A.**

Development of Windows Applications by Means of the MFC Library of Classes in the Microsoft Visual C++2008 Environment of Programming: Textbook. — Dubna: JINR, 2010. — 136 p.

ISBN 978-5-9530-0262-2

This manual describes the technology for building applications for the Windows platform based on the MFC library of classes in the Microsoft Visual C++2008 environment. It provides a detailed description of the structure and technology for building applications and their basic components, such as: child and dialogue windows, menu, toolbars,control elements, dynamic link libraries and AppWizard. It also contains examples of applications with detailed comments.

**ISBN 978-5-9530-0262-2**



## Оглавление

# 1. Введение

Разработка современных Windows-приложений только с использованием Windows API-интерфейса (*Application Programming Interfaces*) для большинства задач представляется неэффективной и не может удовлетворить программиста. Это привело к появлению новых подходов в разработке сложного программного обеспечения. С одной стороны, это использование новых *систем визуального программирования* (среды разработки), а с другой – применение *специальных библиотек классов* для взаимодействия с операционной системой Windows (в дальнейшем в тексте просто Windows), которые самостоятельно выполняют всю «черновую работу», структурируя и облегчая процесс создания программного обеспечения.

В данном курсе выбрана среда разработки *Microsoft Visual C++2008* с библиотекой классов MFC (*Microsoft Foundation Classes*). Существуют и другие библиотеки классов для Windows, но преимущество MFC состоит в том, что она специально написана компанией-разработчиком Windows. Популярность библиотеки MFC настолько высока, что не только такие известные фирмы, как *Watcom* и *Symantec*, приобрели на нее лицензии, но и фирма *Borland*, имеющая собственную библиотеку классов *OWL* (*Object Windows Library*), включила в свою версию 5.0 возможность трансляции приложений, созданных с использованием MFC-библиотеки классов.

# 2. Разработка Windows-приложений с использованием библиотеки классов MFC

## 2.1. Основы и структура MFC-приложений

### 2.1.1. Особенности библиотеки классов MFC

Библиотека классов MFC включает в себя практически весь программный API-интерфейс Windows и позволяет использовать при программировании средства более высокого уровня, чем обычные *вызовы* API-функций. Это упрощает разработку Windows-приложений, имеющих сложный интерфейс пользователя, облегчает поддержку технологии *OLE* (*Object Linking and Embedding* – управления и обмена информацией между программным интерфейсом других приложений) и взаимодействие с базами данных. Библиотека классов MFC преследует две основные цели. Во-первых, скрыть от программиста то, что является вспомогательным при написании Windows-программ. В частности, в MFC-программах не нужно переносить из приложения в приложение практически не изменяющуюся функцию *WinMain*, а можно сразу приступить к решению конкретной задачи. Во-вторых, в отличие от *Win32 API*, обеспечить разработчиков средством более структурированным и понятным, т. е. классами и методами их использования, что существенно упрощает создание Windows-приложений.

На сегодняшний день существует более десятка различных версий библиотек классов MFC, в которых содержится все необходимое для написания Windows-приложений. В этих библиотеках одни классы можно использовать непосредственно, а другие – в качестве базовых для создания новых классов. В MFC-приложениях программист очень редко напрямую вызывает API-функции Windows. Вместо этого он создает *объекты классов* MFC и вызывает их *функции-члены*. В библиотеке MFC определены сотни функций-членов, которые служат оболочкой API-функций, и часто их имена совпадают с именами соответствующих API-функций. Например, для изменения местоположения окна в *Win32 API* определена функция *SetWindowPos*. В MFC-программе это действие выполняется с помощью функции-члена *CWnd::SetWindowPos*. По сути, в MFC представлены практически все API-функции Windows. Кроме того, в ней также имеются средства обработки сообщений, диагностики ошибок и другие средства, обычные для Windows приложений.

Современные системы визуального программирования позволяют автоматизировать процесс создания Windows-приложений. Для этого используются специальные *генераторы приложений*. Программист отвечает на вопросы генератора и тем самым определяет требуемые свойства приложения. Например, поддерживает ли оно многооконный режим, технологию *OLE*, трехмерные органы управления, справочную систему и т. д. Генератор создаст приложение, отвечающее его требованиям, и предоставит исходные коды

программы. Пользуясь ими как шаблонами, можно быстро разработать свое приложение.

Аналогичное средство автоматизированного создания приложений имеет компилятор *Microsoft Visual C++*, называемый *MFC AppWizard*. Заполнив несколько диалоговых панелей, можно указать характеристики приложения и получить его исходные коды с обширными комментариями. *MFC AppWizard* позволяет создавать однооконные и многооконные приложения, а также приложения, не имеющие главного окна (вместо него используется диалоговая панель). Можно также включить поддержку технологии *OLE*, баз данных, справочной системы. Возможности *MFC AppWizard* позволяют создать собственный многооконный редактор текста с возможностью сервера и клиента *OLE*, при этом исходный код приложения можно сразу оттранслировать и получить выполняемый модуль, полностью готовый к использованию.

MFC является не просто библиотекой классов, она также предоставляет программисту *каркас приложения* – заготовку, содержащую набор классов и функций для выполнения типичных операций в Windows-приложении. Программист может разрабатывать собственные приложения, перегружая виртуальные функции классов каркаса и добавляя в него новые классы. Центральное место в каркасе приложения MFC занимает *класс-приложение CWinApp*. В нем скрыты самые общие аспекты работы приложения, например, главный цикл обработки сообщений.

В каркасе приложения MFC есть понятия высокого уровня, которых нет в Windows API. Так, например, архитектура *документ/вид* является «мощной» структурой, надстроенной над API и позволяющей отделить данные программы от их графического представления. Эта архитектура отсутствует в *Win32 API* и полностью реализована в каркасе приложения с помощью классов MFC. Так как *MFC AppWizard* создает исходные коды только с использованием библиотеки классов MFC, требуется знать структуру и возможности этой библиотеки.



### 2.1.2. Преимущества библиотеки классов MFC

При создании MFC перед разработчиками стояли две основные задачи: во-первых, она должна служить *объектно-ориентированным интерфейсом* для доступа к API-функциям Windows с помощью повторно используемых *компонент-классов*, и, во-вторых, расходы по времени вычислений и по объему памяти при использовании MFC должны быть минимальны. Для достижения первой цели были разработаны классы, *инкапсулирующие* окна и другие объекты Windows. В этих классах предусмотрено много виртуальных функций, которые можно перегружать в производных классах и тем самым модифицировать поведение объектов. Уменьшение расходов по времени вычислений и памяти было достигнуто за счет способа реализации классов MFC и способа связи между объектами MFC и объектами Windows.

В Windows информация о свойствах и текущем состоянии окна хранится в ее служебной памяти. Эта информация скрыта от приложений, которые работают с окнами исключительно посредством *дескрипторов* (переменных типа *HWND*). В MFC «оболочкой» окна является класс *CWnd*, но в нем нет переменных-членов, дублирующих все свойства окна с заданным *HWND*. В классе *CWnd* хранится только дескриптор окна. Для этого заведена открытая переменная-член *CWnd::m_hWnd* типа *HWND*. Когда программист запрашивает у объекта *CWnd* какое-нибудь свойство окна (например, заголовок), то этот объект вызывает соответствующую API-функцию и затем возвращает полученный результат. Описанная схема применяется в MFC для реализации практически всех классов, служащих оболочками объектов Windows.

Библиотека MFC по сравнению с *Win32 API* обладает следующими преимуществами:

- представленный набор функций и классов отличается логичностью и полнотой;
- открывает доступ ко всем используемым API-функциям, включая функции управления окнами приложений, сообщениям, элементам управления, меню, диалоговым окнам, объектам *GDI* (*Graphics Device Interface* –

интерфейс графических устройств), таким как шрифты, кисти, перья и растровые изображения, функции работы с документами и многое другое;

- является готовым каркасом приложения, устроенным таким образом, что объекты Windows (окна, диалоговые окна, элементы управления и др.) выглядят в программах как объекты классов C++;
- содержит средства автоматического управления сообщениями;
- устраняет необходимость в организации цикла обработки сообщений (основного источника ошибок в API-приложениях) – вместо стандартного блока *switch/case* все сообщения связываются с функциями-членами, выполняющими соответствующую обработку;
- получаемый программный код достаточно эффективен, а скорость выполнения приложений, основанных на MFC, будет примерно такой же, как и скорость выполнения приложений, написанных с использованием стандартных API-функций Windows;
- обеспечивает динамическое определение типов объектов, что позволяет отложить проверку типа динамически созданного объекта до момента выполнения программы (информация о типе объекта возвращается во время выполнения программы, поэтому программист освобождается от целого этапа работы, связанного с типизацией объектов).

Кроме того, важной особенностью MFC является то, что она может «сосуществовать» с API-приложениями, т. е. в одной и той же программе программист может использовать классы MFC и вызывать API-функции Windows. Такая прозрачность среды достигается за счет согласованности программных обозначений в обеих архитектурах (файлы заголовков, типы и глобальные константы MFC не конфликтуют с именами из API Windows) и согласованности механизмов управления памятью.

### 2.1.3. Архитектура *документ/вид* каркаса MFC-приложения

В устройстве каркаса MFC-приложения важнейшую роль играет архитектура *документ/вид*. Это такой способ проектирования приложения, когда в нем отдельно создаются объекты *документ,* ответственные за хранение данных, и объекты *вид,* ответственные за отображение этих данных. Базовыми классами для объектов *документ* и объектов *вид* в MFC служат классы *CDocument* и *CView*. Классы каркаса приложения *CWinApp*, *CFrameWnd* работают совместно с *CDocument* и *CView*, чтобы обеспечить функционирование приложения в целом. MFC-приложения можно создавать и без использования объектов *документ* и *вид*, но доступ к большинству возможностей каркаса возможен только при поддержке архитектуры *документ/вид*.

Преимуществами применения архитектуры *документ/вид* в MFC-приложении являются: значительное упрощение печати и ее предварительного просмотра; готовый механизм сохранения и чтения документов с диска; преобразование приложений в серверы документов *ActiveX* (приложения, документы которых можно открывать в *Internet Explorer*).

#### а) Объект документ

В MFC-приложении данные программы хранятся в объекте *документ*. Объект *документ* – это объект класса, унаследованного от *CDocument*, и служит для абстрактного представления данных в программе, которые должны быть отделены от их визуального представления. На самом деле это просто некоторая структура данных, которая может описывать что-либо, например, колоду карт в карточной игре, имена и пароли пользователей в сетевой программе и т. д. Обычно у объекта *документ* есть открытые функции-члены, с помощью которых другие объекты, в первую очередь объекты *вид,* могут обращаться к данным документа. Вся обработка данных выполняется только объектом *документ*. Данные документа хранятся в виде переменных-членов подкласса *CDocument*. Их можно сделать открытыми, но, в целях защиты, переменные-члены лучше задавать защищенными и завести для доступа к данным специальные функции-члены. Например, в текстовом редакторе объект *документ* может хранить символы в виде объекта *CByteArray* и предоставлять доступ к ним с помощью функций-членов *AddChar* и *RemoveChar*.

В MFC-классе *CDocument* есть несколько виртуальных функций, позволяющих настроить поведение документа в конкретном приложении. Некоторые из них практически всегда перегружаются в подклассах *CDocument* (табл.1). В приложении MFC создает объект *документ* только один раз при запуске приложения, а при открытии или закрытии файлов этот объект используется повторно. Для этого используются две виртуальные функции *OnNewDocument* и *OnOpenDocument*. Функция *OnNewDocument* вызывается при создании нового документа в программе (*Файл → Создать*), а *OnOpenDocument* – при загрузке документа с диска (*Файл → Открыть*).

Таблица 1. Виртуальные функции *CDocument*

| | |
|---|---|
| *OnNewDocument* | Вызывается при создании нового документа, перегружается для выполнения инициализации, необходимой при создании каждого нового документа |
| *OnOpenDocument* | Вызывается каркасом при загрузке документа из файла |
| *DeleteContents* | Вызывается для удаления содержимого документа, перегружается для освобождения памяти и других ресурсов, выделенных объекту *документ* |
| *Serialize* | Вызывается для записи/чтения данных документа из файла, перегружается почти всегда, чтобы документы можно было хранить в файлах |

### b) Объект вид

Объект *вид* в MFC выполняет две задачи: генерирует визуальное представление документа на экране и преобразует сообщения от пользователя в команды, влияющие на данные документа. Следовательно, объекты *документ* и *вид* тесно взаимосвязаны, и между ними происходит двунаправленный обмен информацией. В MFC-приложении основные свойства объекта *вид* определены в классе *CView*. Кроме того, определен набор подклассов *CView*, расширяющих его функциональные возможности, например, в *CScrollView* добавлены возможности прокрутки *окна-вида*.

В MFC-приложении с объектом *документ* может быть связано любое количество объектов вида, но каждый объект *вид* принадлежит единствен-



ному объекту *документ*. Каркас приложения хранит указатель на объект *документ* в переменной-члене *m_pDocument* у каждого объекта *вид*. Для доступа к этому указателю у объекта *вид* есть функция-член *GetDocument*. Объект *документ* может перебрать все связанные с ним *виды*, просматривая список функциями *GetFirstViewPosition* и *GetNextView*, а объект *вид* может получить указатель на свой документ простым вызовом *GetDocument*.

Таблица 2. Виртуальные функции *CView*

| | |
|---|---|
| *OnDraw* | Вызывается для рисования данных документа внутри объекта *вид* |
| *OnInitialUpdate* | Вызывается при присоединении объекта *вид* к объекту *документ*, перегружается для инициализации объекта *вид* при загрузке объекта *документ* из файла или при создании нового документа |
| *OnUpdate* | Вызывается при любом изменении данных документа, когда необходимо перерисовать *окно-вид*, перегружается для реализации обновления объекта *вид*, когда перерисовывается не все окно, а только некоторая часть |

В классе *CView* (как и в классе *CDocument*) есть несколько виртуальных функций (табл.2) для настройки поведения конкретного объекта *вид*. Самой важной функцией является *OnDraw*, которая вызывается объектом *вид* при получении сообщения *WM_PAINT*. В приложениях, не поддерживающих архитектуру *документ/вид*, сообщения *WM_PAINT* обрабатываются в обработчиках *OnPaint*, а рисование выполняется посредством объектов *CPaintDC*. В приложениях со структурой *документ/вид* сообщение *WM_PAINT* обрабатывается каркасом приложения. В этом обработчике создается объект *CPaintDC* и вызывается виртуальная функция *OnDraw* объекта *вид*. Например, для вывода в центре *окна-вида* строки, хранящейся в объекте *документ*, функция *OnDraw* может быть реализована так:

```
void CProgramView::OnDraw(CDC* /*pDC*/) {
CProgramDoc* pDoc = GetDocument();
if (!pDoc) return;
CDC* pDC = GetDC();
CRect rect;
```

```
GetClientRect(&rect);
pDC->DrawText(pDoc->string, rect, DT_SINGLELINE | DT_CENTER | DT_VCENTER);
}
```

Необходимо обратить внимание на то, что метод *OnDraw* использует контекст устройства, переданный в функцию в качестве параметра, а не создает собственный контекст. Это связано с тем, что каркас приложения использует одну и ту же функцию *OnDraw* для вывода в окно, для печати, и для предварительного просмотра перед печатью. В зависимости от выбранной пользователем команды каркас приложения передает в *OnDraw* различные контексты устройства.

Объект *вид* (как и объект *документ*) в приложении создается только один раз и затем многократно используется. Поэтому в приложениях функция *OnInitialUpdate* класса *CView* вызывается каждый раз, когда документ создается или открывается с диска. По умолчанию *OnInitialUpdate* вызывает функцию *OnUpdate*, а она объявляет все объекты *виды* недействительными, требующим перерисовки. В функции *OnInitialUpdate* удобно поместить инициализацию переменных-членов объекта *вид*, а также другие операции инициализации, необходимые при заведении нового объекта *документ*. Например, в подклассах *CScrollView* в *OnInitialUpdate* обычно вызывается функция-член *SetScrollSizes* для задания границ полос прокрутки. Функция *OnUpdate* вызывается, когда происходит изменение данных документа, а также когда что-нибудь (документ или один из видов) вызывает функцию документа *UpdateAllView*. Функция *OnUpdate* иногда перегружается для ускорения перерисовки с учетом границ областей, связанных с изменившимися данными объекта *документ*.

В приложениях объектов *вид* может быть несколько, но только один из них является активным. Фокус ввода принадлежит активному виду. Для отслеживания, когда объект *вид* становится активным или неактивным, в нем можно перегрузить функцию *CView*::*OnActivateView*. Можно также получить указатель на активный вид или сделать какой-либо вид активным, используя *CFrameWnd*::*GetActiveView* и *CFrameWnd*::*SetActiveView*.



### 2.1.4. Иерархия классов MFC-библиотеки

Библиотека классов MFC содержит большое количество различных классов. Так, например, MFC версии 4.0 включает около 200 классов. Каждый класс содержит от нескольких единиц до нескольких десятков различных методов и элементов данных. Но программисту вовсе не обязательно знать, как устроены все 200 классов, так как большинство приложений можно создать на основе только нескольких основных классов. Мы не станем приводить всю иерархию MFC-классов (ее можно изучить, воспользовавшись документацией или справочной системой среды Visual C++). В данном разделе мы кратко рассмотрим назначение основных классов MFC и их связь друг с другом.

#### а) Базовый класс CObject

Подавляющее большинство классов библиотеки MFC наследовано от базового класса *CObject,* лежащего в основе всей иерархии классов этой библиотеки. Класс *CObject*, а также все классы, наследованные от него, обеспечивают три важных возможности: сериализации, динамического получения информации о классе, диагностической и отладочной поддержки.

Под *сериализацией* подразумевается преобразование данных объекта в последовательную форму, пригодную для записи или чтения из файла.

*Динамическая информация о классе RTCI* (*Run-time class information*) позволяет получить во время выполнения программы название класса и другую важную информацию об объекте.

*Диагностические и отладочные возможности* позволяют проверять состояние объектов подклассов *CObject* на выполнение некоторых условий корректности и выдавать «*дампы*» состояния объектов в отладочное окно Visual C++. Коме того, класс *CObject* предоставляет подклассам ряд полезных дополнительных возможностей. Например, для защиты от утечек памяти в отладочном режиме в классе перегружены операторы *new* и *delete*. Если вы динамически создали объект класса *CObject* и забыли удалить его до завершения программы, то компилятор выдаст в отладочное окно *Visual C++* предупреждающее сообщение.

### b) Класс CCmdTarget

Непосредственно от класса *CObject* наследуется ряд классов, которые сами являются базовыми для остальных классов MFC. В первую очередь это класс *CCmdTarget*, представляющий основу структуры любого приложения. Главной особенностью класса *CCmdTarget* и классов, наследованных от него, является то, что объекты этих классов могут получать от операционной системы сообщения и обрабатывать их. Структура классов, связанных с классом *CCmdTarget*, показана на рис.1.

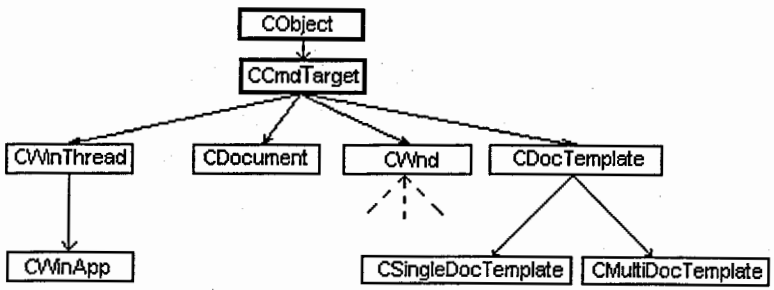


Рис.1. Структура классов, связанных с классом *CCmdTarget*

### c) Классы CWinThread и CWinApp

От класса *CCmdTarget* наследуется класс *CWinThread*, представляющий подзадачи приложения. Простые MFC-приложения, которые мы будем рассматривать в данном курсе, имеют только одну подзадачу. Эта подзадача, называемая главной, представляется классом *CWinApp*, наследованным от класса *CWinThread*.

### d) Класс CDocument

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

### e) Классы CDocTemplate, CSingleDocTemplate и CMultiDocTemplate

Еще один важный класс, наследуемый от *CCmdTarget*, называется *CDocTemplate*. От этого класса наследуются два класса *CSingleDocTemplate* и



*CMultiDocTemplate*. Все эти классы предназначены для синхронизации и управления основными объектами, представляющими приложение – окнами, документами и используемыми ими ресурсами.

### f) Класс CWnd (класс Окна)

Практически все приложения имеют *пользовательский интерфейс*, выполненный на основе окон. Это может быть диалоговая панель, одно или несколько окон, связанных вместе. Основные свойства окон представлены классом *CWnd*, наследованным от класса *CCmdTarget*. На практике редко приходится создавать объекты класса *CWnd*. Класс *CWnd* сам является базовым классом для большого количества классов, представляющих разнообразные окна. Перечислим классы, наследованные от класса *CWnd*:

1. Класс *CFrameWnd* представляет окна, выступающие в роли обрамляющих окон (*frame window*), в том числе главные окна приложения. От этого класса также наследуются классы *CMDIChildWnd* и *CMDIFrameWnd*, используемые для отображения окон многооконного интерфейса MDI.
2. Класс *CMDIFrameWnd* представляет главное окно приложения MDI, а класс *CMDIChildWnd* – его дочерние окна MDI.
3. Класс *CMiniFrameWnd* применяется для отображения окон уменьшенного размера (используются для отображения в них панели управления).
4. Класс *CView* и классы, наследованные от него, представляют окна просмотра *документов* и используются для вывода на экран документа, с которым работает приложение. Разрабатывая приложение, программист наследует собственные классы просмотра документов либо от базового класса *CView*, либо от порожденных классов, определенных в библиотеке MFC. Классы, наследованные от *CCtrlView*, используют для отображения готовых органов управления. Класс *CEditView* использует орган управления *edit*. В классе *CScrollView* определены специальные методы, управляющие полосами просмотра. Класс *CFormView* позволяет создать окно просмотра документа, основанное на диалоговой панели. Классы *CRecordView* и *CDaoRecordView* используются для просмотра записей баз данных.

### g) Класс CDC (контекст отображения)

Для отображения информации в окне или на любом другом устройстве приложение должно получить так называемый *контекст отображения*. Основные свойства контекста отображения определены в классе *CDC*, который непосредственно наследуется от класса *CObject*. От класса *CDC* наследуется четыре различных класса, представляющие контекст отображения различных устройств: *CClientDC, MetaFileDC, CPaintDC, CWindowDC.*

Класс *CClientDC* используется для контекста отображения, связанного с клиентской областью окна (для получения контекста конструктор класса вызывает функцию программного интерфейса *GetDC*, а деструктор функцию *ReleaseDC*). Класс *MetaFileDC* предназначен для работы с метафайлами. Класс *CPaintDC* используется только в обработчике сообщения *WM_PAINT*, когда требуется переопределить функцию *OnPaint* для конкретного дисплея (конструктор класса *CPaintDC* определен так, что выполняет все действия, необходимые для инициализации данного дисплея).

Класс *CWindowDC* позволяет рисовать в произвольной области окна (полноэкранный контекст устройства), для получения контекста конструктор класса вызывает функцию программного интерфейса *GetWindowDC*, а деструктор – функцию *ReleaseDC.*

### h) Класс CGdiObject (объекты графического интерфейса)

Для отображения информации используются различные объекты графического интерфейса – *GDI-объекты*. Для каждого из этих объектов библиотека MFC содержит описывающий его класс, наследованный от базового класса *CGdiObject* (табл.3).

Таблица 3. Краткое описание классов, наследованных от *CDC*

| Класс | Описание |
|---|---|
| *CBitmap* | Растровое изображение *bitmap* |
| *CBrush* | Кисть |
| *CFont* | Шрифт |
| *CPalette* | Палитра цветов |
| *CPen* | Перо |



### i) Класс CMenu (Меню)

Практически каждое приложение имеет собственное меню, которое отображается в верхней части главного окна приложения. Для управления меню в состав MFC включен специальный класс *CMenu*, наследованный от базового класса *CObject*. Для управления меню и панелями управления используется также класс *CCmdUI*. Этот класс не наследуется от базового класса *CObject*. Объекты класса *CCmdUI* создаются, когда пользователь выбирает строку меню или нажимает кнопки управления. Методы класса *CCmdUI* позволяют управлять строками меню и кнопками панели управления.

### j) Вспомогательные функции каркаса приложения

В MFC есть набор *функций-утилит*, существующих независимо от каких-либо классов. Они называются *функциями каркаса приложения*, а их имена начинаются с *Afx*. Функции-члены классов можно вызывать только применительно к объектам этих классов, а функции каркаса приложения можно вызывать из любого места программы. В табл.4 приведены наиболее часто используемые функции каркаса приложения. Они полезны, когда вы хотите вызвать функцию-член или обратиться к переменным этих объектов, но не знаете указателя на них. Функция *AfxGetInstanceHandle* позволяет получить дескриптор экземпляра *EXE-файла* для передачи его функции Windows API (в программах MFC иногда тоже приходится вызывать функции API).

Таблица 4. Функции-члены каркаса MFC-приложения

| Имя функции | Описание |
|---|---|
| *AfxAbort* | Безусловное завершение работы (при ошибке) |
| *AfxBeginThread* | Создает новый поток и начинает его исполнение |
| *AfxEndThread* | Завершает текущий исполняемый поток |
| *AfxMessageBox* | Выводит информационное окно |
| *AfxGetApp* | Возвращает указатель на *объект-приложение* |
| *AfxGetAppName* | Возвращает имя приложения |
| *AfxGetMainWnd* | Возвращает указатель на *главное окно приложения* |
| *AfxGetInstanceHandle* | Возвращает дескриптор экземпляра *EXE-файла* |
| *AfxRegisterWndClass* | Регистрирует оконный класс *WndClass* приложения |

## 2.2. Маршрутизация командных сообщений в MFC-приложении

В библиотеке классов MFC для обработки сообщений используется специальный механизм, который получил название *Message Map* – таблица сообщений. Она состоит из набора специальных макрокоманд, ограниченных макрокомандами *BEGIN_MESSAGE_MAP* и *END_MESSAGE_MAP*, а между ними расположены макрокоманды, отвечающие за обработку отдельных сообщений (например, макрокоманда *ON_COMMAND*).

Макрокоманда *BEGIN_MESSAGE_MAP* представляет собой заголовок таблицы сообщений (например, *BEGIN_MESSAGE_MAP(CEarth_newView, CView)*) и имеет два параметра. Первый параметр содержит имя класса таблицы сообщений, второй указывает его базовый класс. Если в таблице сообщений класса отсутствует обработчик для сообщения, оно передается на обработку базовому классу, указанному вторым параметром этой макрокоманды. Если таблица сообщений базового класса также не содержит обработчика этого сообщения, оно передается следующему базовому классу и т. д. В случае, если ни один из базовых классов не может обработать сообщение, то выполняется обработка по умолчанию, зависящая от типа сообщения.

Макрокоманда *ON_COMMAND* предназначена для обработки командных сообщений. Командные сообщения поступают от меню, кнопок панели управления и клавиш акселераторов. Характерной их особенностью является то, что с ними связан *идентификатор* сообщения. Макрокоманда *ON_COMMAND* имеет два параметра. Первый параметр соответствует идентификатору командного сообщения, а второй – имени метода, предназначенного для обработки этого сообщения. Таблица сообщений должна содержать не больше одной макрокоманды для командного сообщения.

Важной особенностью архитектуры *документ/вид* является то, что в ней можно обрабатывать командные сообщения в любом месте приложения. *Объект главного окна* является основным получателем большинства командных сообщений, но их также можно обрабатывать в *окне вида*, в *окне-документе* и даже в *объекте-приложения*. Для этого надо только добавить соответствующие записи в *карту сообщений* этого класса. Маршрутизация



команд позволяет помещать командные обработчики там, где их разумнее разместить по структуре приложения, а не собирать все обработчики в классе *объекта главного окна*. Обработчики обновления для команд меню, панелей инструментов и других компонент пользовательского интерфейса также включены в механизм маршрутизации. Методы, предназначенные для обработки данного класса сообщений, должны быть определены с ключевым словом *afx_msg* и иметь один параметр – указатель на объект класса *CCmdUI*.

Когда необходимо создать собственные обработчики сообщений, важно помнить, что маршрутизация выполняется только для командных сообщений и для обработчиков обновления. Остальные сообщения Windows, например *WM_CHAR*, *WM_LBUTTONDOWN*, *WM_CREATE* или *WM_SIZE*, должны обрабатываться в окне-получателе сообщения. Например, сообщения от мыши и клавиатуры поступают в объект *вид*, а большинство остальных сообщений обрабатываются в *объекте главного окна*. Объект *документ* и объект *приложение* не получают никаких сообщений кроме командных.

# 3. Разработка MFC-приложений в среде MS Visual C++ 2008

## 3.1. Работа с мастером Application Wizard в MFC-приложении

При разработке Windows-приложения с использованием библиотеки MFC используются четыре основных инструмента:

- *мастер приложений* (*Application Wizard*), генерирующий начальный базовый код разрабатываемой программы (код программы создается автоматически);
- *контекстное меню проекта*, которое применяется *в Class View* для добавления новых классов и ресурсов к вашему проекту (*контекстное меню* отображается при щелчке правой кнопкой мыши в *Class View*, а новый класс создается выбором пункта *Add_Class* в этом меню);
- *контекстное меню класса*, которое используется в *Class View* для расширения и настройки существующих классов программы (для добавления функции в существующий класс применяется опция *Add Function*, а для добавления переменной – *Add Variable*);

- *редактор ресурсов*, применяемый для создания или модификации таких объектов, как меню и панели инструментов, а также «неисполняемых» данных (графических изображений, пиктограмм, меню и диалоговых окон). В среде *MS Visual C++2008* доступно несколько редакторов ресурсов, которые применяются в определенной конкретной ситуации, в зависимости от вида ресурса, требующего редактирования.

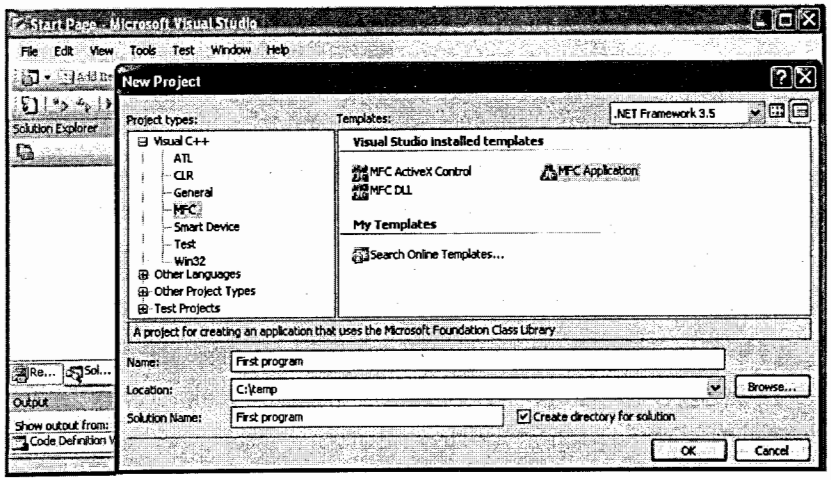


Рис.2. Создание проекта *First program*

С помощью *Application Wizard* создадим MFC Windows-приложение и на этом примере рассмотрим все основные опции, которые потребуется выбрать в процессе его автоматизированного создания. Процесс создания нового MFC-проекта в *Visual C++2008* начинается с выбора пункта *Меню → File→ NewProject.* После этого отображается новое диалоговое окно «*New Project*» (рис.2), в котором в качестве *типа проекта* выберем параметр *MFC*, а в качестве используемого *шаблона приложения – MFC Application.* В этом окне в каталоге «*Name*» требуется ввести имя проекта, которое может быть любым, а в каталоге «*Location*» – путь к этому проекту. Для нашего первого Window-приложения в каталоге «*Name*» запишем имя проекта *First program*, который будет создан в директории *c:\temp*.



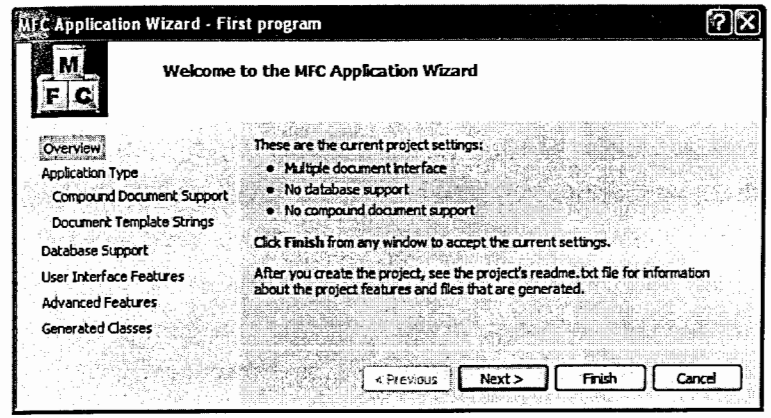


Рис.3. Диалоговое окно *MFC Application Wizard*

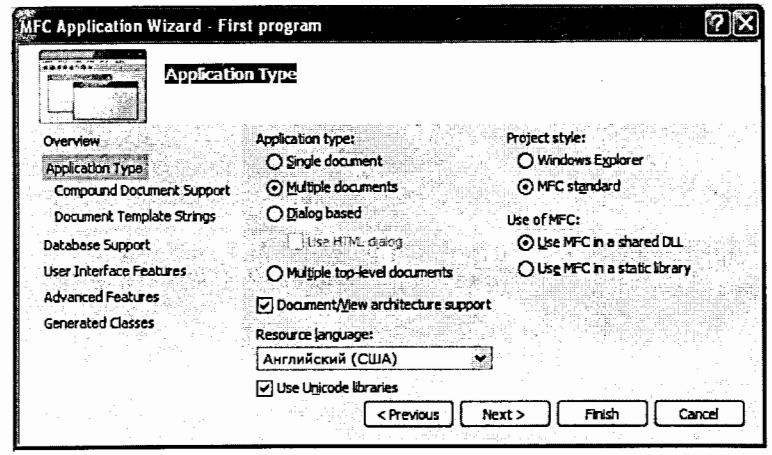


Рис.4. Выбор в качестве типа приложения *Single document*

После щелчка на кнопку *OK* появится диалоговое окно *MFC Application Wizard* – мастер создания приложений MFC. В этом окне пользователь может выбрать требуемые опции для разрабатываемого приложения (рис.3), а мастер *Application Wizard* предоставит ему полный набор дальнейших уточняющих настроек, позволяющих получить заданный тип приложения.

Рассмотрим детально все этапы работы с мастером *Application Wizard.*

1. При выборе в диалоговом окне *Application Type* (рис.4) *Application Wizard* позволяет создать пользователю:

- *SDI-приложение* (*Single Document Interface*) – однодокументный интерфейс, поддерживаемый библиотекой MFC, для программ, в которых нужно открывать только по одному документу в приложении;
- *MDI-приложение* (*Multiple Document Interface*) – многодокументный интерфейс, в котором наряду с возможностью открывать множество документов одного типа, существует возможность организовать одновременную обработку документов разного типа, причем каждый документ отображается в своем собственном окне;
- *Dialog based приложение* для разработки приложения на основе диалогового окна (окном приложения является диалоговое окно, а не обрамляющее окно программы);
- *Multiple top-level documents* (множество документов верхнего уровня) для отображения документов в дочерних окнах рабочего стола, а не в дочерних окнах приложения (как это происходит в *MDI-приложении*);
- *Документ-представление* для поддержки *Document/View architecture*;
- *Resource language* (язык ресурсов) для отображения списка выбора языков, применяемых в ресурсах данного приложения, таких как меню и текстовые строки;
- *Use Unicode libraries* (использовать библиотеки *Unicode*) позволяет поддерживать *Unicode* версии библиотек MFC (если пользователь желает использовать их, то необходимо включить данный флажок);
- *Выбор стиля проекта* между *Windows Explorer* (проводник Windows) и *MFC standard* (первый вариант реализует окно приложения с клиентской областью разделенной на две панели. В левой отображаются данные в форме дерева, а в правой – стандартный текст. Второй вариант использует код MFC для разрабатываемой программы);
- *Выбор типа работы с* MFC-библиотекой. В *Application Wizard* по умолчанию применяется библиотека MFC как разделяемая DLL (динамически подключаемая библиотека). При этом выборе создаваемая программа компонуется с процедурами библиотеки MFC во время ее выполнения. Это позволяет уменьшить размер исполняемого файла, но требует наличия MFC-



библиотеки на компьютере. Если выбрана статическая компоновка, процедуры библиотеки MFC включаются в исполняемый модуль рабочей программы при ее сборке.

В нашем проекте выберем *SDI-приложение* с поддержкой архитектуры *Document/View* и *разделяемой библиотекой DLL*. Кроме того, уберем флажок *Use Unicode libraries*. Если его оставить, то приложение будет ожидать ввода в кодировке *Unicode*, а в файлах будут сохраняться символы *Unicode*, что сделает их нечитаемыми для программ, ожидающих текста *ASCII*.

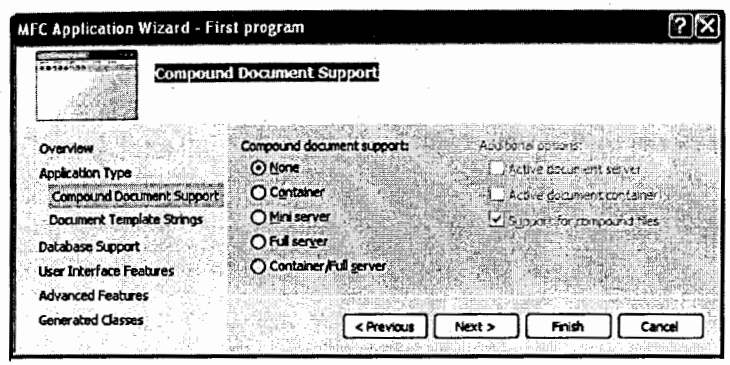


Рис.5. Выбор опции *Compound Document Support в Application Wizard*

2. При выборе в диалоговом окне опции *Compound Document Support Application Wizard* позволяет создать пользователю тип *OLE-приложения* (*контейнер* или *сервер*), а также задать другие опции, относящиеся к технологии *OLE* или *ActiveX* (рис.5). Эти технологии предназначены для создания объектов, распределенных между приложениями. Они включают в себя создание *контейнеров* и *серверов OLE*, реализующих вставку объектов *OLE* в документы приложений-контейнеров (например, редактор формул в *MS Word*). В настоящее время *Microsoft* использует *ActiveX* в качестве основного механизма взаимодействия прикладных программ с системными службами Windows, так как новые механизмы Windows (например, *DirectX*) не поддерживают *API-интерфейс*.

3. При выборе в диалоговом окне опции *Document Template Strings Application Wizard* позволяет пользователю вводить расширение файлов, которые

создает программа, например расширение *txt* или *cpp*. Здесь же в диалоговом окне *Filter Name* можно ввести *имя фильтра*, который будет использоваться в диалоговых окнах *Open* и *Save_As* для фильтрации файлов (в программе будут отображаться только файлы с указанным расширением).

4. Установки в пятом окне *Database Support* мастера *Application Wizard* выполняются пользователем только в том случае, если в приложении осуществляется работа с базами данных. Поэтому мы оставим эту опцию без изменения (*None*) и перейдем к следующему окну *Application Wizard.*

Таблица 5. Дополнительные опции *User Interface Feature*

| Опция | Описание |
|---|---|
| *Thick Frame* (толстая рамка) | Позволяет изменять размер окна приложения, его границу (выбрана по умолчанию) |
| *Minimize box* (кнопка сворачивания) | Предоставляет кнопку сворачивания (выбрана по умолчанию) |
| *Maximize box* (кнопка разворачивания) | Предоставляет кнопку разворачивания (выбрана по умолчанию |
| *Minimized* (свернутое окно) | Если выбрана эта опция, приложение стартует с окном, свернутым в пиктограмму |
| *Maximized* (развернутое) | Если выбрана эта опция, окно приложения при запуске будет развернуто на весь экран |
| *Initial status bar* (начальная панель состояния) | Добавляет панель состояния в нижнюю часть окна приложения, включающую индикаторы клавиш *CAPS LOCK*, *NUM LOCK* и *SCROLL LOCK*, а также строку сообщений |
| *Split window* (разделить окно) | Предоставляет разделительную черту для каждого из основных представлений приложения |
| *Standard docking toolbar* (стандартная стыкуемая панель инструментов) | Добавляет в окно приложения панель инструментов со стандартным набором кнопок, являющихся альтернативой стандартным пунктам меню. Панель инструментов предоставляется по умолчанию. Стыкуемая панель инструментов может быть прикреплена к любой грани окна приложения |
| *Browser style toolbar* (в стиле браузера) | Добавляет в окно приложения панель инструментов в стиле браузера *Internet Explorer* |



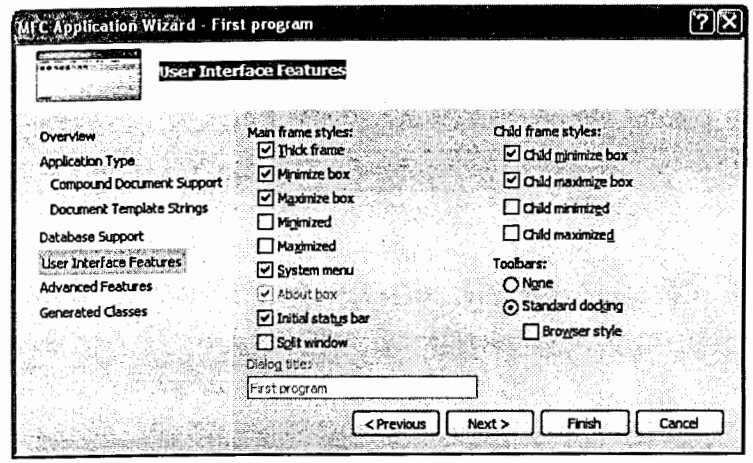


Рис.6. Выбор опции *User Interface Feature в Application Wizard*

5. При выборе в диалоговом окне опции *User Interface Feature* (рис.6) (возможность интерфейса пользователя) из списка в правой панели окна *Application Wizard*, пользователь получает набор опций (табл.5), которые он может включить в разрабатываемое приложение.

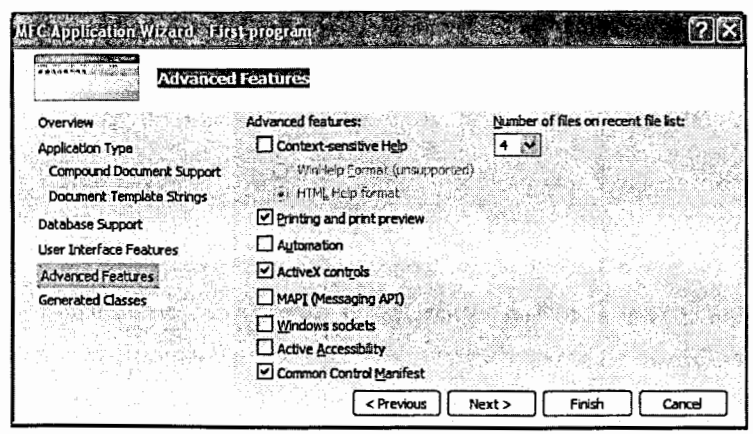


Рис.7. Выбор опции *Advanced Features в Application Wizard*

6. При выборе в диалоговом окне опции *Advanced Features* мастер предоставляет пользователю дополнительные возможности (рис.7). Одна из них – *Printing and print preview* (печать и предварительный просмотр),

которая выбрана по умолчанию, а другая – *Context-sensitive help* (контекстно-чувствительная справка), которую можно получить, если отметить соответствующий флажок. Установка *Printing and print preview* добавляет стандартные пункты *Page Setup* (параметры страницы), *Print Preview* (предварительный просмотр) и *Print* (печать) в меню *File*, а мастер *Application Wizard* предоставит код для их поддержки. Включение опции *Context-sensitive help* предоставляет базовый набор возможностей поддержки контекстной подсказки. В нашем приложении мы оставим эти опции выбранными *по умолчанию* и перейдем к последнему окну *Application Wizard.*

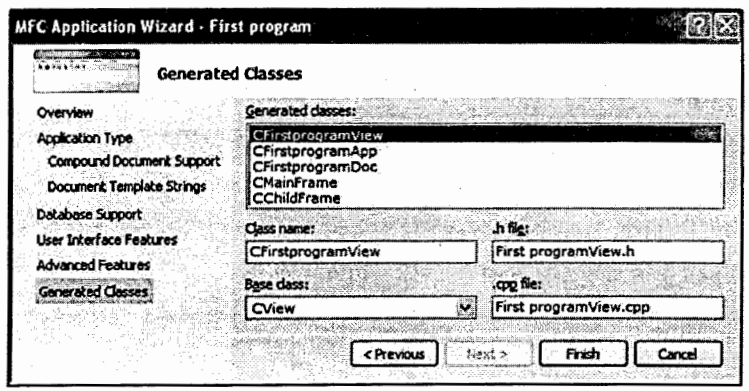


Рис.8. Выбор опции *Generated Classes в Application Wizard*

7. Если в диалоговом окне *Application Wizard* выбрать опцию *Generated Classes*, мастер покажет список классов (рис.8), которые он сгенерирует в коде создаваемой программы. В случае выбора класса *CFirstprogramApp* в проекте можно изменить только имя класса. Если выбрать классы *CFirstprogramDoc* или *CMainFrame*, то можно изменить все, за исключением базового класса. Однако если выбрать класс *CFirstprogramView*, то можно изменить также и базовый класс, который можно выбрать, щелкнув на кнопке со стрелкой вниз (рис.8), для отображения списка классов. В этом списке можно выделить любой класс, и в находящихся ниже полях будет отображено: *имя класса*, *имя заголовочного файла*, в котором находится его определение, *имя базового класса*, а также *имя файла*, содержащего реализацию функций-членов класса.



Определение класса всегда содержится в файле **.h*, а исходный код функций-членов – в файле **.cpp*. Базовые классы и их возможности приведены в табл.6.

Таблица 6. Базовые классы и возможности

| Базовый класс | Возможности класса представления |
|---|---|
| *CEditView* | Обеспечивает редактирование многострочных текстов |
| *CFormView* | Обеспечивает представление формы (диалоговое окно, с элементами управления, управляющими отображением и вводом данных) |
| *CHtmlEditView* | Расширяет класс *CHtmlView* и добавляет возможность редактирования *HTML*-страниц |
| *CHtmlView* | Обеспечивает представление *Web-страницы* |
| *CListView* | Позволяет использовать архитектуру *документ-вид* со списочными элементами управления |
| *CRichEditView* | Обеспечивает редактирование текстовых документов |
| *CScrollView* | Представляет линейки прокрутки, когда отображаемые данные этого требуют |
| *CTreeView* | Обеспечивает возможность использовать архитектуру *документ-вид* с древовидными элементами управления |
| *CView* | Обеспечивает возможности просмотра документов |

Возможности, получаемые в классе вида, зависят от выбранного базового класса. В нашем приложении выберем в качестве базового класса *CView*, затем щелкнем на кнопке «*Finish*», чтобы получить исходные файлы рабочей базовой программы, сгенерированные мастером *Application Wizard* на основе выбранных ранее опций. Все программные файлы, сгенерированные мастером, сохраняются в папке проекта «*First program*», вложенной в *папку решения*, с тем же именем. Кроме того, в папке *res*, вложенной в *папку решения*, будут находиться файлы ресурсов.

## 3.2. Структура MFC-приложения, созданного с помощью мастера *Application Wizard*

### 3.2.1. Просмотр файлов проекта MFC-приложения

При создании приложения в среде *Visual C++* 2008 программисту не нужно беспокоиться о том, какие классы должны присутствовать в его про-

грамме и какие базовые классы использовать в их определении. Все это выполняет мастер установки. На рис.9 показана структура и базовые классы нашего приложения, созданного при помощи *Application Wizard*. Приложение состоит из четырех основных частей: *объект-главного окна* (базовый класс *CmainFrame*), *объект-приложение* (базовый класс *CFirstProgrammApp*), *объект-документ* (базовый класс *CFirstProgrammDoc*), *объект-вид* (базовый класс *CFirstProgrammWiew*).

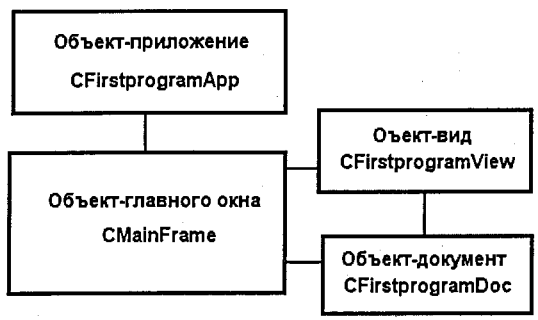


Рис.9. Объекты приложения, созданные *Application Wizard*

В *Visual C++2008* поддерживается несколько способов просмотра информации, имеющей отношение к созданному проекту. Если выбрать вкладку *Solution Explorer* и развернуть его список щелчком на «+» рядом с *TextEditor*, а затем на каждой из папок *Source Files*, *Header Files* и *Resource Files*, то увидим полный список всех файлов проекта, как это показано на рис.10. Просмотреть содержимое любого файла можно, выполнив двойной щелчок на его имени. Содержимое выбранного файла отображается в окне редактора.

В нашем проекте *Application Wizard* создал следующие файлы:

- *Firstprogram.h* – главный заголовочный файл приложения, включает другие заголовочные файлы (в том числе и ресурсные) и объявляет класс приложения *CFirstprogramApp*;
- *Firstprogram.cpp* – главный файл с исходным текстом приложения, содержащий класс *CFirstprogramApp*;
- *MainFrm.h* и *MainFrm.cpp* – файлы содержат класс главного окна приложения *CMainFrame*, производный от базового *CFrameWnd*;



- *FirstprogramDoc.h* и *FirstprogramDoc.cpp* – заголовочный и исходный файлы объекта *документ*;
- *FirstprogramView.h* и *FirstprogramView.cpp* – заголовочный и исходный файлы объекта *вид*;
- *StdAfx.h* и *StdAfx.cpp* – заголовочный и исходный файлы стандартного каркаса приложения;
- *Resource.h*, *Firstprogram.rc*, *Res* – файлы с ресурсными константами, с ресурсами и каталог ресурсов;
- *Firstprogram.dsw* – основной файл рабочей области;
- *Firstprogram.ncb* – файл с информацией о взаимных связях.

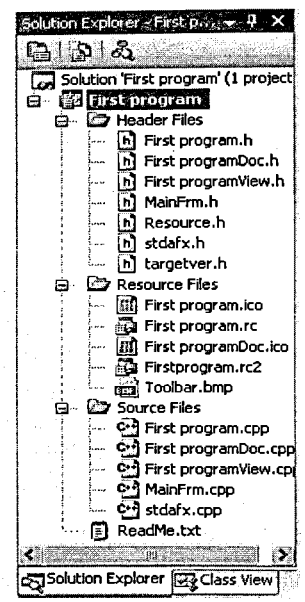


Рис.10. Список всех файлов проекта *First program*

### 3.2.2. Объект приложение

Объект *приложение* является центральной частью в MFC-приложении и содержит главный класс приложения *CFirstprogramApp*. Методы этого класса управляют работой всей программы, выполняют инициализацию приложения, обработку цикла сообщений, объявление и определение пере-

менных и констант. Класс *CFirstprogramAp* объявляется в глобальной области видимости (в файле *Firstprogram.h*), чтобы создание объекта производилось сразу же после запуска приложения. При запуске программы операционная система вызывает функцию *WinMain* объекта *приложение*, которая размещает на экране главное окно Windows-приложения.

```
// First program.h : главный заголовочный файл для приложения
#pragma once
#ifndef __AFXWIN_H__
#error "включите 'stdafx.h' перед включением этого файла для PCH"
#endif
#include "resource.h"        // главные символы
// реализация этого класса CFirstprogramApp:
class CFirstprogramApp : public CWinApp {
public:
        CFirstprogramApp();
// Переопределения
public:
        virtual BOOL InitInstance();
// Реализация
        afx_msg void OnAppAbout();
        DECLARE_MESSAGE_MAP()
};
        extern CFirstprogramApp theApp
```

Основную работу по инициализации выполняет метод *InitInstance* главного класса приложения, определенный в файле *First program.cpp*. Он выполняет необходимую инициализацию при каждом запуске программы, а возвращаемое им значение в методе является признаком удачной или неудачной инициализации. При неудачной инициализации (когда возвращаемое значение *FALSE*) приложение будет закрыто.

// Метод *InitInstance*, файл *First program.cpp*

```
BOOL CFirstprogramApp::InitInstance() {
/* InitCommonControlsEx() требуется под Windows XP, если манифест приложения указывает на использование ComCtl32.dll версии 6 для разрешения визуальных стилей. В противном случае создание окна завершится неудачей. */
➧       INITCOMMONCONTROLSEX InitCtrls;
```



```
        InitCtrls.dwSize = sizeof(InitCtrls);
/* Установите это для включения всех классов общих элементов управления, которые хотите использовать в своем приложении */
        InitCtrls.dwICC = ICC_WIN95_CLASSES;
        InitCommonControlsEx(&InitCtrls);
        CWinApp::InitInstance();
// Инициализировать библиотеки OLE
        if (!AfxOleInit()) {
                AfxMessageBox(IDP_OLE_INIT_FAILED);
                return FALSE;
        }
        AfxEnableControlContainer();
/* Строка, переданная функции SetRegistryKey(), используется для определения ключа реестра, под которым сохраняется информация программы. Под этим ключом сохраняются все настройки приложения, включая список последних использованных программой файлов. Можно изменить ключ реестра */
// TODO: вы должны модифицировать
➧       SetRegistryKey(_T("Local AppWizard-Generated Applications"));
// загрузим файл конфигурации из стандартного INI-файла (включая MRU).
➧       LoadStdProfileSettings(4);
/* Создадим шаблон документа приложения. Объект шаблона документа создается динамически внутри метода InitInstance(). Шаблоны документов служат в качестве соединений между документами, обрамляющими окнами и представлениями */
➧       CSingleDocTemplate* pDocTemplate;
➧       pDocTemplate = new CSingleDocTemplate(
        IDR_MAINFRAME,
        RUNTIME_CLASS(CFirstprogramDoc),
// Создадим главное обрамляющее окно SDI
➧       RUNTIME_CLASS(CMainFrame),
        RUNTIME_CLASS(CFirstprogramView));
        if (!pDocTemplate)
        return FALSE;
// Регистрируем шаблоны документов приложения.
➧       AddDocTemplate(pDocTemplate);
// Выполним стандартную обработку командной строки приложения
➧       CCommandLineInfo cmdInfo;
➧       ParseCommandLine(cmdInfo);
/* Обработаем командную строку приложения. Вернуть FALSE, если приложение было запущено с /RegServer, /Register, /Unregserver или /Unregister */
        if (!ProcessShellCommand(cmdInfo))
```

```
    retum FALSE;
// Было инициализировано единственное окно, покажем и обновим его
➡   m_pMainWnd->ShowWindow(SW_SHOW);
➡   m_pMainWnd->UpdateWindow();
    retum TRUE;
}
```

Рассмотрим процесс создания шаблона документа приложения в методе *InitInstance*. Вначале создается указатель *pDocTemplate* на объекты класса шаблона объектов документов. Для однооконных приложений это класс *CSingleDocTemplate*, а для многооконных – *CMultiDocTemplate*. Объект *шаблона документа* создается динамически внутри метода *InitInstance* оператором *pDocTemplate* = new *CSingleDocTemplate*. Первый параметр в конструкторе *CSingleDocTemplate* – символ, параметр *IDR_MAINFRAME* определяет меню и панель инструментов, используемую типом документа. Следующие три параметра определяют главное обрамляющее окно и объекты класса представления, которые должны быть связаны вместе внутри шаблона документа. Поскольку в нашем проекте мы создаем *SDI-приложение*, здесь присутствует только одно представление, управляемое через объект шаблона документа. *RUNTIME_CLASS* – макрос, позволяющий определять тип объекта класса во время выполнения.

После создания шаблона документа создается главное обрамляющее окно *SDI-приложения*. Для его создания формируется объект класса *CMainFrame,* а в класс *pMainFrame* записывается указатель на него. Класс *CMainFrame* определен в нашем приложении, и мы рассмотрим его немного позже. Затем для только что созданного объекта вызывается метод *LoadFrame* класса *CFrameWnd*, который создает окно, загружает ресурсы, указанные первым параметром, и связывает их с объектом класса *CMainFrame*. Указатель на главное окно приложения записывается в элемент данных *m_pMainWnd* главного класса приложения. Элемент данных *m_pMainWnd* определен в классе *CWinThread*. Когда окно, представленное указателем *m_pMainWnd*, закроется, приложение автоматически будет завершено.



После создания главного окна метод *InitInstance* выводит его с помощью функций-членов *ShowWindow*(*SW_SHOW*) и *UpdateWindow*(). Виртуальные функции *ShowWindow* и *UpdateWindow* унаследованы от базового класса *CWnd* для всех оконных классов MFC, в том числе и для *CFrameWnd*. Функция *ShowWindow* в качестве параметра принимает код состояния окна: *свернутое*, *развернутое* или *обычное* по умолчанию *SW_SHOWNORMAL*. Переменная *m_pMainWnd* содержит указатель на объект главного окна, а оператор «→» действует, как и оператор точка (.), только он работает с указателем на объект (обращается к переменным и методам того объекта, на который ссылается данный указатель). Имена переменных класса в Visual C++ имеют префикс «m_».

Важной особенностью MFC-приложений (в отличие от *Win32 API* приложений) является то, что в исходном тексте программ отсутствует исполняемый код за пределами классов. В приложении нет ни функции *Main*, ни *WinMain*. Единственный оператор за пределами классов – это оператор создания *объекта-приложения* в глобальной области видимости. Чтобы понять, где начинается исполнение программы, разберем структуру каркаса MFC-приложения. В одном из исходных файлов MFC-библиотеки (они поставляются в пакете Visual C++), в *Winmain.cpp*, находится функция *AfxWinMain*. Она является аналогом *WinMain* в MFC-приложениях, и из нее вызываются функции-члены *объекта-приложения*. Отсюда становится ясно, почему объект должен быть *глобальным*, так как глобальные переменные и объекты создаются до исполнения какого-либо кода, а *объект-приложение* должен быть создан до начала исполнения функции *AfxWinMain*.

После запуска программы *AfxWinMain*, для инициализации каркаса приложения, вызывается функция *AfxWinInit*, которая копирует полученные от Windows значения *hInstance*, *nCmdShow* и другие параметры *AfxWinMain* в переменные-члены *объекта-приложения*. Затем вызываются функции-члены *InitApplication* и *InitInstance*. Если какая-либо из функций *AfxWinInit*, *InitApplication* или *InitInstance*, возвращает *FALSE*, то *AfxWinMain* закрывает приложение. При условии успешного выполнения всех перечисленных

функций *AfxWinMain* выполняет следующий, важный шаг: у *объекта-приложения* вызывается функция-член *Run*, т. е. выполняется вход в цикл обработки сообщений главного окна приложения *pApp → Run*. Цикл обработки сообщений завершится при получении из очереди сообщения *WM_QUIT*. Тогда *Run* вызовет функцию *ExitInstance* и вернет управление в *AfxWinMain*, а она выполнит освобождение служебных ресурсов каркаса и затем оператором *RETURN* завершит работу приложения.

Таким образом, всю работу приложения можно разделить на четыре шага: создание *объекта приложения*; выполнение функции *WinMain*, которую предоставляет MFC; вызов с помощью *WinMain* метода *InitInstance* для создания шаблона *документа*, *главного обрамляющего окна*, объектов *документ* и *вид*; вызов функции *Run*, которая выполняет главный цикл сообщений программы, получая и обрабатывая сообщения Windows.

### 3.2.3. *Объект главного окна*

Объект *главного окна* (обрамляющее окно *SDI-приложения*) или «*окно-рамка*» создается объектом класса *CMainFrame*, который является базовым оконным подклассом класса *CFrameWnd*. Этот класс и его потомки предоставляют объектно-ориентированный интерфейс для работы со всеми окнами, создаваемыми в MFC-приложении. Объект *главного окна* отвечает за всю работу программы, которая происходит вокруг его клиентской области. Он отвечает за заголовок, меню, панель инструментов и строку состояния в нижней части окна. Кроме того, он реализует значительную часть функций пользовательского интерфейса в архитектуре *документ/вид*. Например, в классе *CFrameWnd* есть обработчики *OnClose* и *OnQueryEndSession*, которые дают пользователю возможность записать несохраненный документ перед завершением приложения или перед закрытием Windows. Для управления главным окном приложения используется класс *CMainFrame*, определенный в файле *MainFrm.h*:

```
    // MainFrm.h
#pragma once
class CMainFrame : public CFrameWnd {
```



```
protected: // создавать только из сериализации
    CMainFrame();
    DECLARE_DYNCREATE(CMainFrame)
// Атрибуты
public:
// Операции
public:
// Переопределения
public:
    virtual BOOL  PreCreateWindow(CREATESTRUCT& cs);
// Реализация
public:
    virtual ~CMainFrame();
#ifdef _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif
protected:    // встроенные члены управляющий панелей
    CStatusBar   m_wndStatusBar;
    CToolBar     m_wndToolBar;
// Сгенерированные функции отображения сообщений
protected:
    afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
    DECLARE_MESSAGE_MAP()
};
```

Класс *CMainFrame* включает в себя защищенные данные-члены – *m_wndStatusBar* и *m_wndToolBar*, которые являются экземплярами MFC-классов *CStatusBar* и *CToolBar* соответственно. Эти объекты создают и управляют панелью состояния, появляющейся в нижней части окна приложения, и панелью инструментов, предлагающей кнопки для быстрого доступа к стандартным функциям меню. В этом классе есть также функции-члены для получения активного объекта *документ* и объекта *вид* и другие.

### 3.2.4. Объект вид

Объект *вид* создается объектом класса *CFirstprogramView*, который наследуется от базового оконного класса *CView*. Объект *вид* предназначен для работы пользователя с клиентской областью. Клиентская область – это

рабочая зона программы (где она рисует, выводит информацию и т. д.). Объект *вид* отражает в клиентской области окна данные, хранящиеся в объекте *документ*. Например, в клиентскую область (на экран монитора) можно вывести какое-то текстовое сообщение или графический объект.

Определение класса *CFirstprogramView* выполнено в файле *First programView.h*:

```
    // First programView.h
class CFirstprogramView : public CView {
protected: // создавать только из сериализации
    CFirstprogramView();
    DECLARE_DYNCREATE(CFirstprogramView)
// Атрибуты
public:
    CFirstprogramDoc* GetDocument() const;
// Операции
public:
// Переопределения
public:
    virtual void OnDraw(CDC* pDC);
    virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
protected:
    virtual BOOL OnPreparePrinting(CPrintInfo* pInfo);
    virtual void OnBeginPrinting(CDC* pDC, CPrintInfo* pInfo);
    virtual void OnEndPrinting(CDC* pDC, CPrintInfo* pInfo);
// Реализация
public:
    virtual ~CFirstprogramView();
#ifdef _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif
protected:
// Сгенерированные функции отображения сообщений
protected:
    DECLARE_MESSAGE_MAP()
};
#ifndef _DEBUG  // debug version in First programView.cpp
inline CFirstprogramDoc* CFirstprogramView::GetDocument() const
```



```
{ return reinterpret_cast<CFirstprogramDoc*>(m_pDocument); }
#endif
```

Исходный код программы объекта *вид* выполнен в файле *First programView.cpp*:

```
// First programView.cpp:  реализация класса CFirstprogramView
#include "stdafx.h"
#include "First program.h"
#include "First programDoc.h"
#include "First programView.h"
#ifdef _DEBUG
#define new DEBUG_NEW
#endif
// Класс CFirstprogramView
IMPLEMENT_DYNCREATE(CFirstprogramView, CView)
BEGIN_MESSAGE_MAP(CFirstprogramView, CView)
// Стандартные команды печати
    ON_COMMAND(ID_FILE_PRINT, &CView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_DIRECT, &CView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_PREVIEW, &CView::OnFilePrintPreview)
END_MESSAGE_MAP()
// конструктор/деструктор класса CFirstprogramView
CFirstprogramView::CFirstprogramView() {
    // TODO: add construction code here
}
CFirstprogramView::~CFirstprogramView() {
    // TODO: add construction code here
}
BOOL CFirstprogramView::PreCreateWindow(CREATESTRUCT& cs) {
    // TODO: Modify the Window class or styles here by modifying
    //  the CREATESTRUCT cs
    return CView::PreCreateWindow(cs);
}
// CFirstprogramView drawing
void CFirstprogramView::OnDraw(CDC* /*pDC*/) {
CFirstprogramDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    if (!pDoc)
    return;
}
```

```
// CFirstprogramView printing
BOOL CFirstprogramView::OnPreparePrinting(CPrintInfo* pInfo) {
        // default preparation
        return DoPreparePrinting(pInfo);
}
void CFirstprogramView::OnBeginPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/) {
        // TODO: add extra initialization before printing
}
void CFirstprogramView::OnEndPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/) {
        // TODO: add cleanup after printing
}
// CFirstprogramView diagnostics
#ifdef _DEBUG
void CFirstprogramView::AssertValid() const {
        CView::AssertValid();
}
void CFirstprogramView::Dump(CDumpContext& dc) const {
        CView::Dump(dc);
}
CFirstprogramDoc* CFirstprogramView::GetDocument() const // non-debug version
➡    is inline {
        ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CFirstprogramDoc)));
        return (CFirstprogramDoc*)m_pDocument;
}
#endif //_DEBUG
```

Рассмотрим более детально структуру объекта *вид*. Из определения класса *CFirstprogramView* (файл *First programView.h*) видно, что он наследуется от базового класса *CView*. Однако программист может наследовать этот класс и от некоторых других классов библиотеки MFC. Таблица сообщений *BEGIN_MESSAGE_MAP(CFirstprogramView, CView)* в объекте *вид* располагается в начале файла *First programView.cpp*. Непосредственно перед таблицей расположена макрокоманда *IMPLEMENT_DYNCREATE (CFirstprogramView, CView)*, обозначающая, что объекты в этом классе создаются динамически.

Конструктор и деструктор класса *CFirstprogramView* не выполняют полезной работы. *MFC AppWizard* создает для них только пустые шаблоны, которые пользователь может «наполнить» необходимым содержанием.



Таблица 7. Назначение полей структуры *CREATESTRUCT*

| Поле структуры *CREATESTRUCT* | Описание |
|---|---|
| *lpCreateParams* | Указатель на данные, используемые при создании окна |
| *hInstance* | Идентификатор приложения |
| *hMenu* | Идентификатор меню |
| *hwndParent* | Идентификатор родительского окна. Содержит *NULL* |
| *cy* | Высота окна |
| *cx* | Ширина окна |
| *y* | Определяет *y*-координату верхнего левого угла окна. Для дочерних окон координаты задаются относительно родительского окна. Для родительского окна – указываются в экранной системе координат |
| *x* | Определяет *x*-координату верхнего левого угла окна |
| *style* | Стиль класса |
| *lpszName* | Указатель на строку, закрытую двоичным нулем, в которой находится имя окна |
| *lpszClass* | Имя класса окна |
| *dwExStyle* | Дополнительные стили окна |

Виртуальный метод *PreCreateWindow* определен в классе *CWnd*. Он вызывается непосредственно перед созданием окна, связанного с объектом класса. В качестве параметра *cs* ему передается структура *CREATESTRUCT*, определяющая характеристики создаваемого окна. Пользователь может изменить данные, записанные в этой структуре, чтобы повлиять на внешний вид создаваемого окна. Классы, наследованные от *CWnd*, в том числе *CView* и *CFrameWnd*, переопределяют этот метод, изменяя структуру *cs*. В табл.7 описано назначение полей структуры *CREATESTRUCT*. *MFC AppWizard* не вносит в структуру *cs* никаких изменений и переопределяет этот метод следующим образом:

```
BOOL CFirstprogramView::PreCreateWindow(CREATESTRUCT& cs) {
        // TODO: Modify the Window class or styles here by modifying
        //  the CREATESTRUCT cs
        return CView::PreCreateWindow(cs);
}
```

Метод *OnDraw* класса *CFirstprogramView* первоначально определен в классе *CView* как виртуальный и вызывается, когда приложение должно перерисовать документ в окне просмотра. В качестве параметра *pDC* методу *OnDraw* передается указатель на контекст устройства, используя который надо отобразить документ. В зависимости от ситуации метод *OnDraw* вызывается для: отображения документа в окне просмотра, вывода на печать и предварительного просмотра документа перед печатью. Контекст устройства в каждом случае используется разный. Используя контекст устройства, переданный параметром *pDC*, программист может вызывать различные методы графического интерфейса, чтобы отображать требуемую информацию в окне. В том случае, если внешний вид документа при выводе его на печать должен отличаться от вывода в окно, можно вызвать метод *IsPrinting* контекста устройства, чтобы определить, для чего вызывается *OnDraw*. *MFC AppWizard* переопределяет метод *OnDraw* класса *CView* следующим образом:

```
void CFirstprogramView::OnDraw(CDC* /*pDC*/) {
    CFirstprogramDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    if (!pDoc)
    return;
}
```

Функция *GetDocument* в методе *OnDraw* возвращает указатель *pDoc* на объект *документ*, связанный с данным окном просмотра, и объявляется в разделе атрибутов класса (файл *FirstprogramView.h*) после комментария *Атрибуты*. Если окно просмотра не связано ни с каким документом, то возвращается значение *NULL*. Если произошла ошибка и переменная *m_pDocument* не указывает на документ приложения, то макрокоманда *ASSERT_VALID(pDoc)* прерывает работу приложения и отображает соответствующее сообщение. Метод *GetDocument* имеет две реализации. Одна используется для отладочной версии приложения, а другая – для окончательной. Окончательная версия *GetDocument* определена непосредственно после самого класса окна просмотра *CFirstprogramView* как встраиваемый (*inline*) метод:



```
CFirstprogramDoc* CFirstprogramView::GetDocument() const // non-debug version is
inline {
    ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CFirstprogramDoc)));
    return (CFirstprogramDoc*)m_pDocument;
}
```

Два виртуальных метода *OnPreparePrinting*, *OnBeginPrinting* и *OnEndPrinting*, определенные в классе *CView*, вызываются, если пользователь желает распечатать документ, отображенный в данном окне просмотра.

### 3.2.5. Объект документ

Объект *документ* – это объект класса *CFirstprogramDoc* для работы с данными в программе. Объект *документ* создается динамически шаблоном документа во время работы приложения. В качестве базового класса для него используется класс *CDocument* библиотеки MFC. Для того чтобы объекты любого класса, наследованного от базового класса *CObject* (в том числе и *CFirstprogramDoc*), можно было создавать динамически, необходимо выполнить следующее: в описании класса надо поместить макрокоманду *DECLARE_DYNCREATE*, а в качестве параметра указать имя данного класса; определить без параметров конструктор этого класса; разместить макрокоманду *IMPLEMENT_DYNCREATE* в файле реализации (обычно *MFC AppWizard* размещает эту макрокоманду непосредственно перед таблицей сообщений класса, если, конечно, данный класс обрабатывает сообщения). Макрокоманда *IMPLEMENT_DYNCREATE* имеет два параметра: в первом указывается имя класса, а во втором – имя базового класса.

Заголовочный и исходный файлы объекта *документ* определены в файлах *FirstprogramDoc.h* и *FirstprogramDoc.cpp* соответственно.

```
// First programDoc.h :
#pragma once
class CFirstprogramDoc : public CDocument {
protected: // create from serialization only
    CFirstprogramDoc();
➡   DECLARE_DYNCREATE(CFirstprogramDoc)
    CString string;
```

```
// Атрибуты
        public:
// Операции
        public:
// Переопределения
        public:
        virtual BOOL OnNewDocument();
        virtual void Serialize(CArchive& ar);
// Реализация
        public:
        virtual ~CFirstprogramDoc();
        #ifdef _DEBUG
        virtual void AssertValid() const;
        virtual void Dump(CDumpContext& dc) const;
        #endif
        protected:
// Сгенерированные функции отображения сообщений
        protected:
        DECLARE_MESSAGE_MAP()
        };
```

*// First programDoc.cpp* :

```
#include "stdafx.h"
#include "First program.h"
#include "First programDoc.h"
#ifdef _DEBUG
#define new DEBUG_NEW
#endif
// CFirstprogramDoc
➧       IMPLEMENT_DYNCREATE(CFirstprogramDoc, CDocument)
// таблица сообщений класса
        BEGIN_MESSAGE_MAP(CFirstprogramDoc, CDocument)
        END_MESSAGE_MAP()
// конструктор/деструктор класса CFirstprogramDoc
➧       CFirstprogramDoc::CFirstprogramDoc() {
        // TODO: add one-time construction code here
}
➧       CFirstprogramDoc::~CFirstprogramDoc() {
```



```
}
➧       BOOL CFirstprogramDoc::OnNewDocument() {
        if (!CDocument::OnNewDocument())
                retum FALSE;
        // TODO: add reinitialization code here
        retum TRUE;
}
// CFirstprogramDoc serialization
➧       void CFirstprogramDoc::Serialize(CArchive& ar) {
                if (ar.IsStoring()) { // TODO: add storing code here  }
                Else    { // TODO: add loading code here }
}
// CFirstprogramDoc diagnostics
#ifdef _DEBUG
void CFirstprogramDoc::AssertValid() const {
        CDocument::AssertValid();
}
void CFirstprogramDoc::Dump(CDumpContext& dc) const {
        CDocument::Dump(dc);
}
#endif //_DEBUG
```

После макрокоманды *IMPLEMENT_DYNCREATE* в файле реализации следует таблица сообщений класса *CFirstprogramDoc*, которая не содержит обработчиков сообщений. При разработке приложения программист будет добавлять обработчики различных сообщений к классу *CFirstprogramDoc* и другим классам приложения. Для добавления новых обработчиков сообщений, а также для внесения других изменений в классы мы будем использовать *MFC AppWizard*.

Конструктор и деструктор класса *CFirstprogramDoc*, подготовленный *MFC AppWizard*, содержит пустой блок. В него можно поместить код инициализации для объектов данного класса. Следует иметь в виду, что для приложений, построенных на основе однооконного интерфейса, объект класса документ создается всего один раз. Когда пользователь создает новый документ или открывает документ, уже записанный в файле, используется

старый объект класса, представляющего документ. Вместе с конструктором создается деструктор *~CFirstprogramDoc*. Деструктор не содержит кода и представляет собой такую же заготовку, как и конструктор.

В классе *CFirstprogramDoc* переопределены два виртуальных метода *OnNewDocument* и *Serialize*. Виртуальный метод *OnNewDocument* определен в классе *CDocument*, а вот виртуальный метод *Serialize* определен в классе *CObject*. Поэтому цепочка наследования классов в этом случае длиннее: *CFirstprogramDoc* ← *CDocument* ← *CCmdTarget* ← *CObject*.

Метод *OnNewDocument* вызывается в программе, когда надо создать новый документ для приложения. Если программист хочет переопределить метод *OnNewDocument* (в данном случае за вас это делает *MFC AppWizard*), то сначала необходимо вызвать метод *OnNewDocument* базового класса и только затем можно выполнять инициализацию документа:

```
BOOL CFirstprogramDoc::OnNewDocument() {
    if (!CDocument::OnNewDocument())
    return FALSE;
  // TODO: десь можно выполнить инициализацию документа
    return TRUE;
}
```

Если создание нового документа прошло успешно, метод *OnNewDocument* возвращает значение *TRUE*, а в противном случае *FALSE*. Параметр *FALSE* означает, что создание документа на уровне класса *CDocument* не прошло и следует прекратить дальнейшие действия.

Метод *Serialize* используется в тех случаях, когда надо загрузить документ из файла на диске или, наоборот, записать его в файл. Он вызывается, когда пользователь выбирает из меню *File* строки *Open* или *Save*:

```
void CFirstprogramDoc::Serialize(CArchive& ar) {
    if (ar.IsStoring()) {// TODO: add storing code here }
    Else   {// TODO: add loading code here }
}
```

В качестве параметра *ar* методу *Serialize* передается объект класса *CArchive*, связанный с файлом, в который надо записать или из которого надо прочи-



тать документ. Чтобы узнать, что надо делать с документом, используется метод *IsStoring* класса *CArchive*. Если он возвращает ненулевое значение, значит требуется сохранить состояние документа в файле. В противном случае необходимо считать документ из файла. Более подробно использование метода *Serialize* для сохранения и восстановления документов будет рассмотрено в подпункте 3.7.2.

Класс *CFirstprogramDoc* содержит переопределения еще двух виртуальных методов – *AssertValid* и *Dump*, входящих в базовый класс *CObject*. Описание этих методов и их определение расположено в блоке *#ifdef_DEBUG*, поэтому эти методы используются только для отладочной версии приложения.

## 3.3. Графический интерфейс устройств (GDI) в MFC-приложении

### 3.3.1. Контекст устройства

В Windows существует несколько средств для вывода графической информации на экран, включая *DirectDraw*, *OpenGL*, *GDI* и т. д. В наших MFC-приложениях мы будем использовать *GDI* (*Graphics Device Interface*) – подсистему Windows, ответственную за вывод графики и текста на дисплей и принтер. Именно она занимается выводом большинства «окон», которые пользователь Windows видит на экране. *GDI* является базовым и простейшим способом вывода графики в Windows. При выводе на экран графической информации (линии, текста, изображения) программа приложения обращается к функциям *GDI*. Эти функции поддерживаются каркасом MFC и для удобства пользователя объединены в классы.

Основным классом для работы с графикой является класс *CDC* и производные от него *CPaintDC*, *CClientDC*, *CWindow*, *CMetaFileDC* (рассмотренные в подпункте 2.3.1), которые отличаются от базового класса только конструкторами и деструкторами (исключением является класс *CMetaFileDC*). Класс *CDC* инкапсулирует понятие контекста устройства (*DC – device context*). *Контекст устройства* – это структура данных, содержащая информацию о параметрах и атрибутах вывода графики на устройство, например, дисплей или принтер. Такая информация, в частности, включает в себя: палитру

устройства, определяющую набор доступных цветов; параметры пера для черчения линий; параметры кисти для закраски и заливки; параметры шрифта, использующегося для вывода текста. Контекст устройства является посредником между операционной системой Windows и устройством вывода, тем самым обеспечивается аппаратная независимость программы.

В *GDI* существуют пять типов контекста устройства:

- с дисплеем (*Display DC*) – для вывода на экран;
- с принтером (*Printer DC*) – для печати на принтер;
- контекст виртуального устройства в памяти (*Memory DC*) – для создания растровых изображений в памяти с возможностью быстрого их копирования в другие типы контекстов и обратно;
- контекст метафайла (*Metafile DC*) – для вывода графики в *метафайл* (метафайл – это хранилище последовательности команд *GDI*, каждая из которых описывает одну графическую функцию, создающую результирующий рисунок);
- информационный контекст (*Information DC*).

Первые четыре типа контекста устройства (*Display DC, Printer DC, Memory DC Metafile DC*) предоставляют унифицированный интерфейс для вывода графической информации на разнотипные устройства, освобождая приложение (и разработчика) от необходимости заботиться о том, куда именно производится вывод графики. Информационный контекст для вывода не используется. Он служит только для получения информации о параметрах и поддерживаемых режимах устройств, с которыми связан.

### 3.3.2. Работа приложения с контекстом устройства дисплея

Операционная система Windows поддерживает три типа контекста дисплея: *контекст класса, приватный контекст* и *общий контекст*. Первые два типа используются в приложениях, которые выводят на экран большое количество информации. Примером такого рода приложений являются издательские системы, графические пакеты и т. д. Приложения, которые не работают интенсивно с экраном, используют общий контекст. Контекст класса



является устаревшим и поддерживается только для обеспечения совместимости с предыдущими версиями Windows (*Microsoft* не рекомендует использовать его при разработке новых приложений).

Для построения изображения в окне или для вывода графики в определенное окно приложения требуется воспроизвести последовательность команд:

- с помощью функции *GetDC* получить дескриптор контекста устройства, связанного с окном, в котором вы собираетесь рисовать или выводить графику;
- нарисовать все, что требуется, с помощью функций *GDI*;
- освободить контекст с помощью функции *ReleaseDC*.

### 3.3.3. Создание и уничтожение объектов CDC

Для отображения текста или графики в приложении необходимо создать объект контекста устройства, соответствующий окну или устройству вывода данных. При рисовании этот объект сохраняет выбранные средства и установленные атрибуты и предоставляет функции-члены для рисования точек, линий, прямоугольников и других фигур. В MFC-приложении это выполняется с помощью функции *OnDraw* класса *CDC*, и по умолчанию используется текущий контекст устройства объекта вида, адрес которого передается в эту функцию:

```
void CFirstprogramView::OnDraw(CDC* /*pDC*/) {
➧       // отображение графики, используя указатель pDC→
}
```

Если программа отображает или рисует графику не в окне объекта *вид* (а в каком-то другом, например, в диалоговом), то класс окна для рисования предоставляет обработчик сообщений *WM_PAINT*, называемый *OnPaint*. Он создает объект контекста устройства, порождаемый от класса *CPaintDC*.

Управление созданием и удалением объектов *CDC* является важной частью Windows-приложения. Программа никогда напрямую не обращается к контексту устройства, она обращается к нему через вызовы определенных функций. После того как все действия произведены и необходимость в

использовании устройства отпала, программа должна освободить контекст устройства, чтобы не занимать память. Кроме того, есть еще одна причина, из-за которой необходимо освобождать контекст устройства. В системе может быть одновременно только ограниченное число контекстов устройств. Если контекст устройства не «освобождается» после операций вывода, то через несколько «перерисовок» окна система может «зависнуть».

Рассмотрим два варианта *корректной работы* с объектами контекстов класса *CDC*.

Вариант 1. Создать объект контекста устройства в стеке (тогда он будет уничтожен автоматически):

```
void CFirstprogramView::OnDraw(CDC* /*pDC*/) {
// создадим контекст устройства в стеке
    CClientDC dc(this);
//при помощи функции gdi →TextOutA выведем объект string класса CString
    dc.TextOutA(20,20,pDoc->string);
}
```

Вариант 2. Получить указатель на объект с помощью *CWnd::GetDC*(), при этом перед выходом необходимо вызвать функцию *ReleaseDC*():

```
void CFirstprogramView::OnDraw(CDC* /*pDC*/) {
// получим указатель на текущий контекст устройства
    CDC* pDC = GetDC();
//при помощи функции gdi →TextOutA выведем объект string класса CString
    pDC->TextOutA(20,20,pDoc->string);
// освободим контекст устройства с помощью функции ReleaseDC()
    ReleaseDC(pDC);
}
```

Чтобы вывести в программе графику за пределы *рабочей области окна*, необходимо создать объект класса *CWindowDC*, а для ее отображения в окне объекта *вид* или другом окне из функции, которая не обрабатывает сообщения методов *OnDraw* или *OnPaint*, нужно создать объект контекста устройства класса *CClientDC*, являющегося членом MFC. Например, создание контекста устройства в функции *Graph*():

```
void CFirstprogramView::OnGraph() {
```



```
// для отображения графики используем объект класса CClientDC
// получаем указатель на этот контекст устройства
    CClientDC *pDC = new ClientDC (this)
//при помощи функции gdi →TextOutA выводим объект string класса CString
    pDC->TextOutA(20,20,pDoc->string);
// освобождаем контекст с помощью функции delite pDC;
    delete pDC;
}
```

Так как все функции рисования являются членами класса *CDC* (базового класса по отношению к остальным классам контекста устройства), то эти функции вызываются при использовании объекта контекста устройства любого произвольного типа.

### 3.3.4. Состояние контекста устройства. Объекты GDI

Свойства контекста устройства назначаются с помощью методов класса *CDC*, а его состояние определяется связанными с ним графическими объектами. Объекты *GDI* загружаются в контекст устройства вызовом перегружаемой функции *SelectObject*. В любой текущий момент с контекстом может быть связан только один объект каждого типа. Все *GDI*-объекты в MFC представлены с помощью классов, а *CGdiObject* – базовый класс для объектов, которые являются экземплярами классов наследников.

Перечислим классы наследников базового класса *CGdiObject*:

- *CBitmap* – класс, инкапсулирующий растровые изображения (битовые массивы), используется для отображения картинок, создания кистей;
- *CBrush* – класс, использующийся для закрашивания областей окна;
- *CFont* – полный набор символов алфавита определенного вида и размера (шрифты хранятся на диске как ресурсы);
- *CPallete* – класс преобразования цветов, позволяющий задействовать цветовые возможности устройства, не вызывая конфликта с другими приложениями, работающими с этим же устройством;
- *CPen* – класс инструмента для рисования линий и границ фигур;
- *CRgn* – класс области окна, определяемой прямоугольником, эллипсом или всевозможными их комбинациями.

Для создания объектов вызываются конструкторы соответствующих классов, но для некоторых объектов этого бывает недостаточно. Например, создание *GDI*-объектов типа *CFont* или *CRgn* требует вызова *CreateFont*() или *CreatePolygonRgn*().

Память, выделенная под *GDI*-объекты, принадлежит процессу и освобождается при его завершении. Так как объекты (особенно растровые изображения) могут занимать значительный объем памяти, то требуется оперативное отслеживание их своевременного удаления. Прежде чем удалять *GDI*-объект, его необходимо вначале «отсоединить» от контекста устройства. Например, для удаления пера, созданного в методе *OnDraw* для вывода графики, используется функция *newpen.DeleteObject*. При уничтожении контекста устройства все связанные с ним *GDI*-объекты отсоединяются.

### 3.3.5. Основные функции класса CDC контекста устройства

Для вывода графической информации в классе *CDC* существует набор функций, которые можно разделить на следующие категории: методы рисования линий (*LineTo*, *MoveTo*, *Polyline*, *Arc*, *ArcTo*, *PolyBezier*, и т. д.); методы рисования замкнутых фигур (*Ellipse*, *Rectangle*, *Polygon*, *Pie*, *Chord* и т. д.); методы вывода текста (*TextOut*, *DrawText* и т. д.); функции работы с растровым изображением (*GetPixel*, *SetPixel*, *FloodFill*, *BitBlt* и т. д.). Кроме того, существует отдельная категория функций для работы с контекстом устройства по переключению режимов и установке параметров вывода графической информации. Часть из них устанавливается напрямую через определенные функции (например, *SetBkColor*), а другие устанавливаются с помощью специальных графических объектов. Например, объект *pen* задает режим вывода линий (цвет, толщина, стиль), *brush* регулирует режим закраски фигур (цвет, стиль), *font* задает свойства шрифта, которым выводится текст, *palette* задает набор используемых в *DC* цветов, *region* используется для задания областей отсечения, вне которых вывод графики блокируется.

Рассмотрим эти функции более подробно:



1. *CreateDC*(*LPCTSTR lpszDriverName, LPCTSTR lpszDeviceName, LPCTSTR lpszOutput, const void* lpInitData*) служит для инициализации контекста устройства, где первый параметр – указатель на строку с именем драйвера устройства, второй – указатель на строку с именем устройства (необходим, если драйвер поддерживает несколько устройств), третий – указатель на строку с именем файла или порта, куда будет осуществляться вывод, четвертый содержит особые параметры для настройки данного устройства (эта функция используется редко, обычно MFC сам создает необходимый контекст устройства).
2. *CreateCompatibleDC*(*CDC* pDC*) создает в памяти объект контекста устройства, совместимого с текущим, указатель на который передается в качестве параметра.
3. *SelectObject* служит для связи с контекстом устройства *GDI*-объекта. Например, функция *CPen* SelectObject* (*CPen* pPen* связывает с контекстом устройства перо, указатель на которое передан в качестве параметра, и возвращает указатель на перо, которое находилось в контексте устройства, до вызова функции, а при неудаче возвращает *NULL*.
4. *CPen* GetCurrentPen* возвращает указатель на выбранное в контекст перо (функции с префиксом *Get* возвращают определённые параметры контекста устройства).
5. *SetBkMode* (*int nBkMode*) устанавливает режим закрашивания фона.
6. *COLORREF SetPixel*(*int X, int Y, COLORREF Color*) закрашивает пиксель экранной области с координатами (*X,Y*), переданными в качестве первых двух аргументов цветом *Color* (третий аргумент функции).
7. *Rectangle*(*int upX, int upY, int lowX, int lowY*) рисует прямоугольник пером, загруженным в контекст, и заполняет его кистью, загруженной в контекст. В качестве аргументов в эту функцию передаются координаты противоположных углов.
8. *RoundRect*(*int upX, int upY, int lowX, int lowY, int curveX, int curveY*) рисует и заполняет текущей кистью контекста прямоугольник со скругленными угла-

ми, параметры *curveX* и *curveY* задают ширину и высоту эллипса, определяющего дугу для скругленных углов.

9. *Ellipse*(*int upX, int upY, int lowX, int lowY*) рисует и заполняет текущей кистью контекста эллипс, вписанный в прямоугольник, координаты углов которого передаются в функцию в качестве параметров.

10. *MoveTo*(*CPoint point*) перемещает фокус в точку *point*.

11. *LineTo*(*POINT point*) проводит линию пером, загруженным в контекст устройства, из фокуса в точку, переданную в функцию в качестве параметра.

12. Функции, используемые для преобразования системы координат:

*virtual int SetMapMode*(*int nMapMode*);
*virtual CPoint SetViewportOrg*(*CPoint point*);
*CPoint SetWindowOrg*(*CPoint point*);
*virtual CSize SetViewportExt*(*CSize size*);
*virtual CSize SetWindowExt*(*CSize size*);
*void DPtoLP*(*LPPOINT lpPoints, int nCount = 1*) *const*;
*void LPtoDP*(*LPPOINT lpPoints, int nCount = 1*) *const*.

Создание и удаление объектов производится с помощью соответствующих функций (например, объект pen создается с помощью *CreatePen*, а удаляется с помощью *DeleteObject*). Режимы, задающиеся через графические объекты, переключаются с помощью создания новых объектов и указания контексту *DC* использовать их для вывода графики. Это делается помощью функции *SelectObject*. При выборе нового объекта через *SelectObject* в качестве возвращаемого значения передается дескриптор объекта, который использовался в *DC* до замены.

## 3.4. Создание и вывод графики в Windows MFC-приложении

### 3.4.1. Организация процедуры вывода в контекст устройства дисплея

Для вывода текста или графики в окно Windows-приложения требуется выполнить следующие действия: получить или создать объект нужного контекста устройства; выбрать стандартные или создать специальные инструменты рисования; в полученном контексте что-то нарисовать или вывести текстовую информацию; удалить созданные и используемые инструменты



и объекты; удалить полученный контекст устройства. Рассмотрим эти действия более подробно.

#### a) Работа с объектами контекста устройства

В MFC-приложении, когда программа вызывает контекст устройства, то она получает его с объектами и значениями, уже заполненными по умолчанию. Объект в составе контекста называется текущим объектом. Слово текущий говорит о том, что его можно изменить. В программе можно создать новый объект в контексте устройства, например, *bitmap* или *шрифт*, и сделать его текущим. Замещенный объект из памяти программы автоматически не удаляется, и чтобы не занимать память, его требуется удалять отдельно. Пользователь может получить характеристики любого текущего объекта в контексте устройства, но изменить эти характеристики он может только через замену этого объекта.

Инструменты рисования, определенные в контексте устройства, т. е. перья и кисти, принадлежат к категории объектов, называемых *графическими* или *объектами GDI* (термин объект относится к структуре данных Windows, а не к объекту C++). Существуют и другие графические объекты: шрифты, растровые изображения, области, контуры и палитры. Полное описание этих объектов приведено в разделах справочной системы MFC «*Platform SDK, Graphics and Multimedia Services, GDI*».

#### b) Выбор стандартных инструментов рисования

В операционной системе Windows есть набор предопределенных часто используемых перьев, кистей, шрифтов и других объектов *GDI*, которые не надо создавать в приложении, а можно сразу использовать готовые. Они называются стандартными объектами *GDI* (табл.8). Их можно выбирать в контекст устройства с помощью функции *CDC::SelectStockObject* или присваивать уже существующим объектам *CPen*, *CBrush* и др. с помощью *CGdiObject::CreateStockObject*. Класс *CGdiObject* является базовым классом для *CPen*, *CBrush*, *CFont* и других MFC-классов, представляющих объекты *GDI*.

Таблица 8. Стандартные объекты рисования *GDI*

| Функция | Описание |
|---|---|
| *NULL_PEN* | Пустое (прозрачное) перо |
| *BLACK_PEN* | Черное сплошное перо толщиной в 1 пиксель |
| *WHITE_PEN* | Белое сплошное перо толщиной в 1 пиксель |
| *NULL_BRUSH* | Пустая (прозрачная) кисть |
| *HOLLOW_BRUSH* | То же, что *NULL_BRUSH* |
| *BLACK_BRUSH* | Черная кисть |
| *DKGRAY_BRUSH* | Темно-серая кисть |
| *GRAY_BRUSH* | Серая кисть |
| *LTGRAY_BRUSH* | Светло-серая кисть |
| *WHITE_BRUSH* | Белая кисть |
| *ANSI_FIXED_FONT* | Моноширинный системный шрифт *ANSI* |
| *ANSI_VAR_FONT* | Пропорциональный системный шрифт *ANSI* |
| *SYSTEM_FONT* | Системный шрифт для пунктов меню, элементов управления и т. п. |
| *SYSTEM_FIXED_FONT* | Моноширинный системный шрифт |

В контексте устройства имеются два инструмента класса *CDC* – перо и кисть, выбор которых отражается на работе функции рисования. Перо влияет на способ рисования линии. Оно действует как на прямые и кривые линии, так и на границы замкнутых фигур. Кисть действует только на способ рисования внутренней области замкнутых фигур. При первичном создании контекст устройства содержит заданные по умолчанию перо и кисть. Перо рисует сплошную черную линию шириной в один пиксель независимо от текущего режима отображения (который будет рассмотрен далее). Кисть заливает внутреннюю область фигуры с замкнутым контуром непрозрачным белым цветом. Идентификаторы для этих инструментов выбраны по умолчанию.

Если в программе требуется изменить текущий инструмент рисования на новый, то необходимо выбранный инструмент (табл.8) связать с текущим объектом контекста устройства. Для этого нужно передать идентификатор выбранного инструмента в функцию *SelectStockObject* класса *CDC*: *CGdiObject* SelectStockObject* (*int nIndex*), где *nIndex* – идентификатор объекта, который передается в контекст устройства. Например, выберем в контекст устройства стандартные объекты белое перо и серую кисть:



```
void CFirstprogramView::OnDraw(CDC* /*pDC*/) {
        CDC* pDC = GetDC();
    ➧ pDC->SelectStockObject (WHITE_PEN) ;
    ➧ pDC->SelectStockObject (GRAY_BRUSH) ;
    // Вызов других графических функций и рисование графики
    ➧ ReleaseDC(pDC);
}
```

При выборе пера с параметром *NULL_PEN* линии не рисуются. Аналогично при выборе кисти с параметром *NULL_BRUSH* внутренняя часть фигуры не закрашивается. Этот выбор удобен при рисовании фигур, состоящих только из границ, когда необходимо оставить неизменным существующее на экране графическое изображение внутри границы. Выбранные объекты будут использоваться в программе до следующего выбора других инструментов рисования.

### c) Создание нестандартных инструментов рисования

В контексте устройства можно создать и нестандартные инструменты рисования, например перо или кисть. Для этого в программе необходимо выполнить следующие действия:

- создать объект класса *CPen* для пера или *CBrush* для кисти;
- вызывать соответствующие функции класса *CPen* или *CBrush* для инициализации пера или кисти;
- «связать» перо или кисть с объектом контекста устройства, сохранив указатель на предыдущее перо или кисть;
- вызывать функции рисования для выполнения графического вывода;
- удалить из контекста созданные перо или кисть и возвратить в контекст их старые значения.

Для создания временного пера или кисти объявляется объект класса *CPen* или *CBrush* как локальный объект внутри функции, генерирующей графический вывод. Этот метод продемонстрирован в фрагменте следующей программы:

```
void CFirstprogramView::OnDraw(CDC* /*pDC*/) {
        CDC* pDC = GetDC();
```

```
// объявляем черное и голубое перо класса CPen как локальные объекты
    CPen BlackPen(PS_SOLID, 1, RGB(0, 0, 0));
    CPen BluePen(PS_SOLID, 1, RGB(0, 255, 255));
    CPen *pOldPen;
// объявляем зеленую кисть класса CBrush как локальный объект
    CBrush  GreenBrush(RGB(0, 255, 0));
    CBrush *pOldBrush;
// выбираем голубое перо и рисуем квадрат
    pOldPen = pDC->SelectObject(&BluePen);
    pDC->MoveTo(10, 10);     pDC->LineTo(100, 10);
    pDC->LineTo(100, 100);   pDC->LineTo(10, 100);
    pDC->LineTo(10, 10);
// Возвращаем перо
    pDC->SelectObject(pOldPen);
// выбираем черное перо, рисуем окружность и закрашиваем ее зеленой кистью
    pDC->SelectObject(&BlackPen);
    pOldBrush = pDC->SelectObject(&GreenBrush);
    pDC->Ellipse(10, 10, 100, 100);
// возвращаем старые объекты pen и brush в DC
    pDC->SelectObject(pOldPen);
    pDC->SelectObject(pOldBrush);
// освобождаем контекст устройства DC
    ReleaseDC(pDC);
}
```

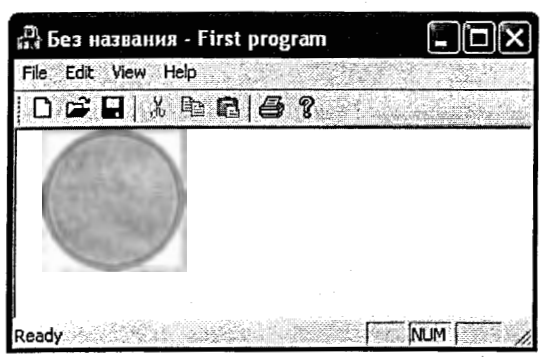


Рис.11. Вывод графических объектов с использованием нестандартных инструментов рисования

На рис.11 показан вид окна приложения *First program*, реализующего работу данного фрагмента программы по выводу графических объектов с использованием нестандартных инструментов рисования.



При инициализации пера или кисти Windows присваивает им дескрипторы и сохраняет их при выборе этих инструментов в объект контекста устройства внутри функции, генерирующей графический вывод. Поэтому при вызове функции *SelectObject* для создания нового пера или кисти нужно сохранить возвращаемый указатель на дескриптор замещаемого объекта:

```
pOldPen     =     pDC -> SelectObject(&BluePen);
pOldBrush   =     pDC -> SelectObject(&GreenBrush);
```

После отображения выводимой информации с использованием созданных объектов рисования необходимо их удалить из контекста устройства, чтобы он не хранил некорректный дескриптор. Это выполняется вызовом функции

```
pDC -> SelectObject(pOldPen);
pDC -> SelectObject(pOldBrush);
```

Функция *CreatePen* (*int nPenStyle, int nWidth, COLORREF crColor*) создает перо с требуемыми параметрами. Параметр *nPenStyle* описывает стиль линии, нарисованной пером. Стиль *PS_NULL* создает перо, совпадающее со стандартным пером *NULL_PEN*. Стиль *PS_INSIDEFRAME* используется для рисования границы вокруг фигуры с замкнутым контуром. Стили *PS_DASH, PS_DOT, PS_DASHDOT* и *PS_DASHDOTDOT* применяются, если ширина пера равна одному пикселю. Если ширина превышает этот размер, то перечисленные стили генерируют сплошные линии. Параметр *nWidth* описывает ширину линии в логических единицах, используемых в текущем режиме отображения. Если ширина пера 0, то ширина линии будет один пиксель независимо режима отображения. Параметр *crColor* задает цветовой код линии, которым легче всего описать цвет, используя макрос *Win32 RGB*. Параметры *bRed*, *bGreen* и *bBlue* показывают относительную интенсивность красного, зеленого и синего цветов. Каждому параметру можно присвоить значение в диапазоне от 0 до 255. В табл.9 приведены параметры 16 стандартных цветов, записанных с помощью функции *RGB*.

Объект *кисть* можно инициализировать так, чтобы она закрашивала внутреннюю область фигур, вызывая функцию *CreateSolidBrush* (*COLORREF crColor*) класса *CBrush* с параметром *crColor*, описывающим цвет заливки.

Кроме того, для заливки внутренней области фигур можно инициализировать кисть, вызвав функцию *CreateHatchBrush*(*int nindex, COLORREF crColor*) класса *Cbrush*. Параметр *nIndex* задает узор. Параметр *crColor* описывает цвет линий штриховки.

Таблица 9. Параметры 16 стандартных *RGB*-цветов

| Красный | Зеленый | Синий | Цвет |
|---|---|---|---|
| 0 | 0 | 0 | Черный |
| 0 | 0 | 255 | Синий |
| 0 | 255 | 0 | Зеленый |
| 0 | 255 | 255 | Бирюзовый |
| 255 | 0 | 0 | Красный |
| 255 | 0 | 255 | Малиновый |
| 255 | 255 | 0 | Желтый |
| 255 | 255 | 255 | Белый |
| 0 | 0 | 128 | Темно-Синий |
| 0 | 128 | 0 | Темно-Зеленый |
| 0 | 128 | 128 | Темно-Бирюзовый |
| 128 | 0 | 0 | Темно-Красный |
| 128 | 0 | 128 | Темно-Малиновый |
| 128 | 128 | 0 | Темно-Желтый |
| 128 | 128 | 128 | Темно-Серый |
| 192 | 192 | 192 | Светло-Серый |

Объект *кисть* можно инициализировать так, чтобы она закрашивала внутреннюю область фигур, вызывая функцию *CreateSolidBrush* (*COLORREF crColor*) класса *CBrush* с параметром *crColor*, описывающим цвет заливки. Кроме того, для заливки внутренней области фигур можно инициализировать кисть, вызвав функцию *CreateHatchBrush*(*int nindex, COLORREF crColor*), класса *Cbrush*. Параметр *nIndex* задает узор. Параметр *crColor* описывает цвет линий штриховки.

### d) Режимы отображения графики

Большинство MFC-функций, работающих с оконными координатами, определяют координаты окна относительно начала рабочей области (от ее левого верхнего угла). При этом единицы, в которых измеряются координаты, зависят от режима отображения (*mapping mode*), установленного для данного



окна. Единицы измерения, зависящие от режима отображения, называют *логическими единицами*, а координаты в этом случае называют *логическими координатами*. При выводе информации на конкретное устройство, например, на экран монитора, единицы логических координат преобразуются в *физические единицы*, которыми в этом случае являются пиксели. В MFC встроены функции для настройки физической (аппаратной) и логической систем координат. В MFC также предусмотрены функции перехода от одной системы к другой (режим преобразования координат).

Режим преобразования координат – это атрибут контекста устройства, задающий способ пересчета логических координат в физические координаты устройства. Логические координаты передаются функциям рисования, а контекст преобразует их в физические координаты пикселов на экранном окне или на листе принтера. Например, если вызывается функция рисования *Rectangle*: *pDC->Rectangle*(*0, 0, 200, 100*), то нельзя сказать, что будет нарисован прямоугольник шириной 200 пикселов и высотой 100 пикселов. Эта функция нарисует прямоугольник шириной 200 логических единиц и высотой 100 логических единиц, так как в режим преобразования координат контекста устройства по умолчанию передается идентификатор *MM_TEXT* (табл.10), где логическая единица равна одному пикселю. В других режимах масштаб может быть иным. Например, в режиме *MM_LOMETRIC* одна логическая единица равна 1/10 мм. Следовательно, в этом режиме при вызове функции *Rectangle* будет нарисован прямоугольник шириной 20 мм и высотой 10 мм. Режимы, отличные от *MM_TEXT*, приведены в табл.10. Их удобно применять для рисования в одинаковом масштабе на различных устройствах вывода.

Задача программиста состоит в том, чтобы определить, когда и какую систему координат использовать. Основные правила при работе с системами координат: все параметры, передаваемые в методы *CDC*, – это логические координаты; все параметры, передаваемые в методы *CWnd*, – это физические или аппаратные координаты. Для значения параметров, сохраняемых длительное время, надо использовать логические координаты.

Таблица 10. Идентификаторы режимов отображения

| Идентификатор | Значение | Эффект |
|---|---|---|
| *MM_TEXT* | 1 | Логическая единица равна пикселю, ось *x* направлена вправо, ось *y* вниз, режим задан по умолчанию |
| *MM_LOMETRIC* | 2 | Логическая единица равна 0,1 мм, ось *x* направлена вправо, ось *y* вверх |
| *MM_HIMETRIC* | 3 | Логическая единица равна 0,01 мм, ось *x* направлена вправо, ось *y* вверх |
| *MM_LOENGLISH* | 4 | Логическая единица равна 0,01 дюйма, ось *x* направлена вправо, ось *y* вверх |
| *MM_HIENGLISH* | 5 | Логическая единица равна 0,001 дюйма, ось x направлена вправо, ось y вверх. |
| *MM_TWIPS* | 6 | Логическая единица равна 1/1440 дюйма, ось *x* направлена вправо, ось *y* вверх |
| *MM_ISOTROPIC* | 7 | Позволяет настраивать размерность осей, их направления и начало отсчета |
| *MM_ANISOTROPIC* | 8 | Позволяет настраивать отдельно каждую ось (размерность направление и начало отсчета) |

Рассмотрим функции для работы с системами координат.

1. Функции для перемещения центров координат: *SetViewportOrg*(*CPoint point*) и *SetWindowOrg*(*CPoint point*). Первая смещает центр физических координат, а вторая – логических в точку, переданную в качестве параметра. Обе функции возвращают координаты предыдущего центра.

2. Функции для задания единиц измерения: *SetViewportExt*(*CSize size*) и *SetWindowExt*(*CSize size*). Первая функция устанавливает единицы измерения физической системы координат, вторая – логической.

3. Функция для перехода от аппаратных координат к логическим *DPtoLP* (*LPPOINT lpPoints, int nCount=1*)*const* и функция для перехода от *логических* к *физическим LPtoDP*(*LPPOINT lpPoints, int nCount=1*)*const*. Аргументами обеих функций являются указатель на массив с точками, которые нужно преобразовать, и размерность этого массива.

4. Функция для установления направления осей и единиц измерения в задаваемой системе координат *SetMapMode*(*int nMapMode*). Возможные



режимы задаются параметром *nMapMode*, который принимает значения в соответствии с табл.10.

5. Функция для настройки начала координат физической области вывода *SetViewportOrg* (*int x, int y*) и функция для настройки начала координат логической области вывода *SetWindowOrg* (*int x, int y*).

Последние четыре функции в качестве параметров получают новые значения для установки начала координат и возвращают предыдущие. Причем для физической области вывода указываются координаты, которые являются началом координат, а для логической области вывода указываются координаты, которые соответствуют левому верхнему углу логической области вывода. Поэтому настраивать начало координатной сетки нужно только для одной из областей, так как система координат логической области вывода привязана к системе координат физической области.

При работе с правыми системами координат удобнее переместить начало координатной системы из левого верхнего угла в другую точку, например, в центр окна. Для этого нужно вызвать одну из двух функций *SetWindowOrg* (смещение левого верхнего угла поверхности изображения) или *SetViewportOrg* (смещение начала логической системы координат). Поясним это на примере. Пусть в программе требуется вывести графический объект эллипс в правой системе координат (рис.12). Для этого надо поместить начало системы координат в центр окна клиентской области и поменять направление оси *y*. Это можно сделать следующим образом:

```
void CFirstprogramView::OnDraw(CDC* /*pDC*/) {
    CDC* pDC = GetDC();
// устанавливаем режим MM_ISOTROPIC для замены направления оси y
    ➧ pDC->SetMapMode(MM_ISOTROPIC);
// устанавливаем единицы измерения физической системы координат
    ➧ pDC->SetWindowExt(100, 100);
// устанавливаем единицы измерения логической системы координат
    ➧ pDC->SetViewportExt(rect.Width(), -rect.Height());
// устанавливаем новое начало системы координат
    ➧ pDC->SetViewportOrg( rect.Width()/2, rect.Height()/2 );
// рисуем оси правой системы координат
```

```
    pDC->MoveTo(0, 0);
    pDC->LineTo(0, 90);
    pDC->MoveTo(0, 0);
    pDC->LineTo(90, 0);
// выводим графический обьект эллипс в новой системе координат
    pDC->Ellipse(10, 10, 30, 30);
    ReleaseDC(pDC);
}
```

Результат работы программы вывода графического объекта эллипс в клиентскую область окна приложения с правой системой координат показан на рис.12.

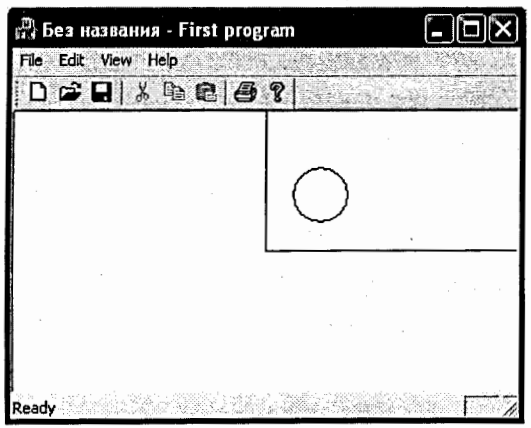


Рис.12. Вывод эллипса в правой системе координат области окна

Иногда в программе требуется узнать характеристики устройства, с которым связан контекст. Для этого вызывается функция *GetDeviceCaps*. Например, получить ширину и высоту экрана можно так:

```
void CFirstprogramView::OnDraw(CDC* /*pDC*/) {
    CDC* pDC = GetDC();
// Объявляем два объекта типа CString
    CString str1,str2;
// Получаем разрешение монитора но горизонтали и вертикали
    int cx = pDC->GetDeviceCaps( HORZRES );
    int cy = pDC->GetDeviceCaps( VERTRES );
// Преобразуем переменные типа int  в String
    str1.Format("%7d",cx); str2.Format("%7d",cy);
// Выводим полученные параметры на экран
```



```
    pDC->TextOutA(10,10,str1); pDC->TextOutA(10,20,str2);
    ReleaseDC(pDC);
}
```

### 3.4.2. Рисование графических примитивов с помощью функций GDI

В этом подпункте рассмотрим создание графики в Windows-приложении с помощью *GDI*-функций, а именно, покажем, как нарисовать простые графические объекты (линии, фигуры и т. д.). За основу нашего приложения возьмем созданный в пункте 3.1 шаблон приложения «*First program*» и внесем в него все необходимые исправления и добавления с тем, чтобы пользователь смог рисовать и выводить данные в окне приложения.

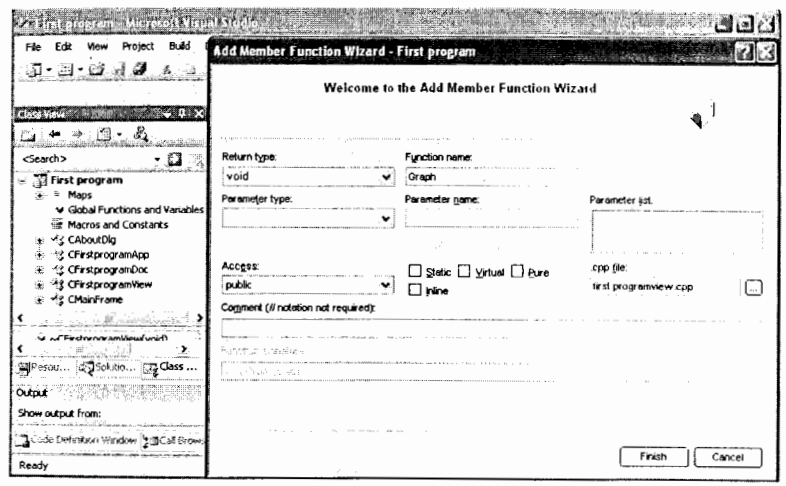


Рис.13. Добавление новой функции *Graph* в класс *CFirstprogramView*

Откроем проект приложения *First program*. Затем в контекстном меню *Class View* выберем опцию *Add Member Function* (рис.13) и добавим в класс объекта вида *CFirstprogramView* новую функцию *Graph* типа *void*. Мастер установки *Application Wizard* включил эту функцию в заголовочный файл *CFirstprogramView.h* и создал исходный код в файле *CFirstprogramView.cpp*. Чтобы вывести стандартный графический объект в функции *Graph*, которая не обрабатывает сообщения метода *OnDraw*, нужно создать объект контекста устройства класса *CDC*:

```
void CFirstprogramView::OnGraph() {
```

```
// получим указатель на объект с помощью CWnd::GetDC()
    ➧ CDC* pDC = GetDC();
// нарисуем графический объект
// освободим контекст устройства
    ➧ ReleaseDC(pDC);
}
```

### a) Рисование линий и отрезков

Для рисования отрезка надо поместить текущую позицию в один из концов отрезка и вызвать функцию *LineTo* с координатами ее второго конца:

```
void CFirstprogramView::OnGraph() {
        CDC* pDC = GetDC();
    ➧ pDC->MoveTo( 40, 60 );        pDC->LineTo( 40, 160 );
      ReleaseDC(pDC);
}
```

При выводе нескольких соединяющихся отрезков достаточно вызвать функцию *MoveTo* только для одного из концов первого отрезка, например:

```
➧ pDC->MoveTo(60, 60 );
➧ pDC->LineTo( 60, 180 );        pDC->LineTo( 100, 180 );
```

Несколько отрезков линии можно построить одним вызовом функций *Polyline* или *PolylineTo*. Отличие между этими функциями состоит в том, что *PolylineTo* пользуется текущей позицией, а *Polyline* – нет. Например, квадрат можно нарисовать так:

```
➧ POINT aPoint[5] = { 80, 40, 180, 40, 180, 140, 80, 140, 80, 40 };
➧ pDC->Polyline( aPoint, 5 );
```

или с помощью *PolylineTo*:

```
➧ pDC->MoveTo( 200, 40 );
➧ POINT bPoint[4] = {300, 40, 300, 140, 200, 140, 200, 40 };
➧ pDC->PolylineTo( bPoint, 4 );
```

### b) Рисование эллипсов, многоугольников и других фигур

В *GDI* есть функции для рисования более сложных объектов, чем отрезки и прямые: *Chord* – рисует замкнутую фигуру, образованную пересечением эллипса и отрезка; *Ellipse* – рисует эллипс или окружность; *Pie* – рисует сектор круговой диаграммы; *Polygon* – рисует многоугольник; *Rectangle* – рисует прямоугольник; *RoundRect* – рисует прямоугольник с закругленными углами.



Функциям *GDI*, рисующим замкнутые фигуры, передаются координаты описывающего прямоугольника. Например, чтобы функцией *Ellipse* нарисовать окружность, надо указать не центр и радиус, а описывающий квадрат, например:

```
➧ pDC->Ellipse( 40, 210, 140, 310 );
```

Координаты описывающего прямоугольника можно передавать в виде структуры *RECT* или как объект *CRect*:

```
➧ CRect rect1( 300, 210, 450, 310 ); pDC->Ellipse( rect1 );
```

Для рисования дуг окружностей и эллипсов предназначена функция *CDC::Arc*. В качестве параметров ей передаются координаты описывающего эллипс прямоугольника и координаты начальной и конечной точек дуги (эти точки задают углы для вырезания дуги из эллипса, поэтому могут точно на него не попадать). Ниже приведен пример для рисования *дуги* эллипса шириной 200 единиц и высотой 100 единиц:

```
➧ CRect rect2( 300, 60, 500, 140 );
➧ CPoint point1(450, 110);
➧ CPoint point2(320, 40);
➧ pDC->Arc(rect2, point1 ,point2 );
```

Для этих графических объектов создадим в контексте устройства инструменты рисования перья и кисти:

```
void CFirstprogramView::Graph() {
        CDC* pDC = GetDC();
        // 1. Создадим сплошное цветное перо:
        CPen BlackPen(PS_SOLID, 1, RGB(0, 0, 0));
        CPen BluePen(PS_SOLID, 1, RGB(0, 255, 255));
        CPen RedPen(PS_SOLID, 1, RGB(255,0 , 0));
        CPen YellowPen(PS_SOLID, 1, RGB(255, 255, 0));
        CPen GreenPen(PS_SOLID, 3, RGB(0, 255, 0));
        CPen *pOldPen;
      // 2. Создадим сплошную кисть:
        CBrush  GrayBrush(RGB(127, 127, 127));
        CBrush  YellowBrush(RGB(255, 255, 0));
      // 3. Создадим штриховую кисть:
        CBrush  GreenBrush(HS_DIAGCROSS, RGB(0, 255, 0));
        CBrush *pOldBrush;
```

```
    // 4. Нарисуем линию черным пером:
      pDC->MoveTo( 40, 60 ); pDC->LineTo( 40, 160 );
      ReleaseDC(pDC);
}
```

### c) Отображение текста

В подпункте 2.1.3 уже упоминался один из способов вывода текста в клиентскую область окна с помощью функции *DrawText* класса *CDC*. При ее применении нужно указать прямоугольник, внутри которого выводить текст, и флаги, указывающие расположение текста внутри него. Например, при выводе текста в виде одной строки по центру прямоугольника *rect* использовался вызов функции в следующем виде:

```
pDC->DrawText(pDoc->string, rect, DT_SINGLELINE | DT_CENTER | DT_VCENTER);
```

Кроме *DrawText*, в классе *CDC* есть еще несколько функций для работы с текстом. Одна из самых часто используемых – функция *TextOut*, которая подобно *DrawText* выводит текстовую строку, а в качестве параметров принимает координаты точки начала вывода текста или использует для этого текущую позицию. Например, *DrawText(0,0,"TEST")* выведет строку *"TEST"*, начиная с левого верхнего угла окна, связанного с контекстом *DC*. Функция *TabbedTextOut* при выводе текстовой строки заменяет символы табуляции на пробелы (позиции табуляции передаются в функцию в качестве параметра).

По умолчанию координаты, переданные в функции *TabbedTextOut, TextOut*, и *ExtTextOut*, считаются левым верхним углом описывающего прямоугольника для первого символа строки. Однако эту интерпретацию координат можно изменить, задав в контексте свойство по выравниванию текста. Это выполняется с помощью функции *SetTextAlign*, например, для выравнивания текста по правой границе: *pDC->SetTextAlign(TA_RIGHT)*.

В качестве примера по выводу текста добавим надписи над каждым графическим объектом из предыдущего фрагмента программы *First program*. Эту операцию выполним с помощью функции *TextOut*. Для этого в объекте *документ* (файл *FirstprogramDoc.h*) объявим объекты типа *string* класса *CString*:

```
class CFirstprogramDoc : public CDocument {
```



```
    protected: // create from serialization only
      CFirstprogramDoc();
      DECLARE_DYNCREATE(CFirstprogramDoc)
➧     CString string1,string2,string3,string4,string5,string6,string7;
```

В конструкторе класса (файл *FirstprogramDoc.cpp*) присвоим им значения:

```
CFirstprogramDoc::CFirstprogramDoc() {
➧     string1 = "линия и оси";
➧     string2 = "Polyline";
➧     string3 = "PolylineTo";
➧     string4 = "Круг и Прямоугольник";
➧     string5 = "Эллипс";
➧     string6 = "Дуга";
➧     string7 = "Квадрат";
```

Для того чтобы вызвать объекты класса *CString* в функции *Graph*, необходимо передать в эту функцию указатель на объект *документ*:

```
void CFirstprogramView::Graph() {
➧   CFirstprogramDoc* pDoc = GetDocument();
      ASSERT_VALID(pDoc);
// вывод объекта string1 в окно с началом первого символа в точке 5, 5
 ➧  pDC->TextOutA(5,5,pDoc->string1);
```

Вид Windows-приложения *First program* с результатом работы по выводу графических объектов и надписей в клиентскую область окна показан на рис.14, а ниже представлен код программы функции *Graph*:

```
void CFirstprogramView::Graph() {
// Получим указатель pDoc на объект документ
➧   CFirstprogramDoc* pDoc = GetDocument();
      ASSERT_VALID(pDoc);
// Вызовем контекст устройства дисплея (получим указатель pDC)
       CDC* pDC = GetDC();
// Создадим инструменты для рисования: перья и кисти:
      CPen BlackPen(PS_SOLID, 1, RGB(0, 0, 0));
      CPen BluePen(PS_SOLID, 1, RGB(0, 255, 255));
      CPen RedPen(PS_SOLID, 1, RGB(255,0 , 0));
      CPen YellowPen(PS_SOLID, 1, RGB(255, 255, 0));
      CPen GreenPen(PS_SOLID, 3, RGB(0, 255, 0));
      CPen *pOldPen;
      CBrush  GreenBrush(HS_DIAGCROSS, RGB(0, 255, 0));
```

```
        CBrush  GrayBrush(RGB(127, 127, 127));
        CBrush  YellowBrush(RGB(255, 255, 0));
        CBrush *pOldBrush;
// Вывод графических объектов в клиентскую область окна:
      //1. Выводим надпись "линия и оси"
        pDC->TextOutA(5,5,pDoc->string1);
      //2. Рисуем линию черным пером
        pOldPen = pDC->SelectObject(&BlackPen);
        pDC->MoveTo( 40, 60 ); pDC->LineTo( 40, 160 );
        pDC->SelectObject(pOldPen);
      //3. Рисуем оси голубым пером
        pOldPen = pDC->SelectObject(&BluePen);
        pDC->MoveTo(60, 60 ); pDC->LineTo( 60, 180 );
        pDC->LineTo( 100, 180 );
        pDC->SelectObject(pOldPen);
      //4. Выводим надписи "Квадрат и Polyline"
        pDC->TextOutA(110,5,pDoc->string7);
        pDC->TextOutA(110,20,pDoc->string2);
      //5. При помощи функции Polyline рисуем красным пером квадрат
        pOldPen = pDC->SelectObject(&RedPen);
        POINT aPoint[5] = { 80, 40, 180, 40, 180, 140, 80, 140, 80, 40 };
        pDC->Polyline( aPoint, 5 );
        pDC->SelectObject(pOldPen);
      //6. Выводим надписи "Квадрат и PolylineTo"
        pDC->TextOutA(220,5,pDoc->string7);
        pDC->TextOutA(220,20,pDoc->string3);
      //7. При помощи функции PolylineTo рисуем желтым пером квадрат
        pOldPen = pDC->SelectObject(&YellowPen);
        pDC->MoveTo( 200, 40 );
        POINT bPoint[4] = {300, 40, 300, 140, 200, 140, 200, 40 };
        pDC->PolylineTo( bPoint, 4 );
        pDC->SelectObject(pOldPen);
      //8. Выводим надпись "Круг и Прямоугольник"
        pDC->TextOutA(100,180,pDoc->string4);
      //9. Рисуем черным пером и зеленой штриховой кистью Эллипс
        pOldPen = pDC->SelectObject(&BlackPen);
        pOldBrush = pDC->SelectObject(&GreenBrush);
        pDC->Ellipse( 40, 210, 140, 310 );
        pDC->SelectObject(pOldBrush);
      //10. Рисуем текущим пером Квадрат и закрашиваем серой кистью
```



```
        pOldBrush = pDC->SelectObject(&GrayBrush);
        pDC->Rectangle(160,210, 280,310);
        pDC->SelectObject(pOldBrush);
      //11. Выводим надпись "Эллипс"
        pDC->TextOutA(350,180,pDoc->string5);
      //12. Рисуем текущим пером Эллипс и закрашиваем желтой кистью
        pOldBrush = pDC->SelectObject(&YellowBrush);
        CRect rect1( 300, 210, 450, 310 );
        pDC->Ellipse( rect1 );
        pDC->SelectObject(pOldPen);
        pDC->SelectObject(pOldBrush);
      //13. Выводим надпись "Дуга "
        pDC->TextOutA(400,15,pDoc->string6);
      //14. рисуем Дугу зеленым пером, толщиной 3
        pOldPen = pDC->SelectObject(&GreenPen);
        CRect rect2( 300, 60, 500, 140 );
        CPoint point1(450, 110);
        CPoint point2(320, 40);
        pDC->Arc(rect2, point1 ,point2 );
// Возвращаем в контекст текущее перо и кисть  и освобождаем контекст устройства
        pDC->SelectObject(pOldPen);
        pDC->SelectObject(pOldBrush);
        pDC->SelectObject(pOldPen);
        ReleaseDC(pDC);
}
```

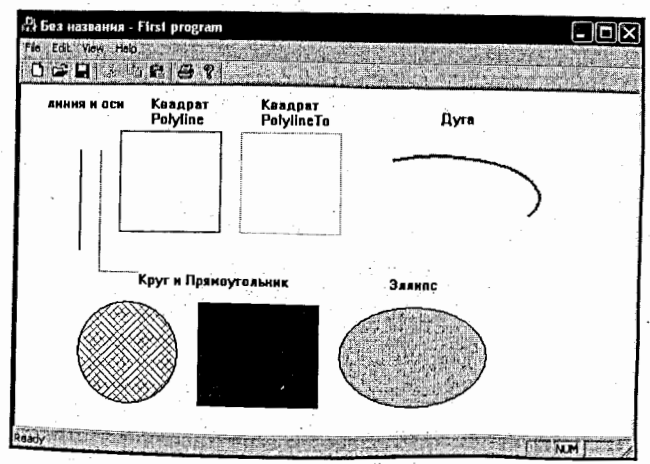


Рис.14. Вывод графических объектов в клиентскую область окна

### 3.4.3. Вывод текста в окно программы

Создадим новое приложение с именем *Text*, которое будет считывать символы с клавиатуры и отображать их в клиентской области окна. Для того чтобы в программе вывести набираемый с клавиатуры текст, надо выполнить следующие шаги:

- подготовить буфер под хранение полученных символов;
- организовать чтение символов с клавиатуры;
- сохранить символы в документе;
- отобразить набранный текст в клиентской области окна.

#### a) Подготовка буфера для хранения символов

Данные в MFC-программах сохраняются в объекте *документ* в виде переменных-членов подкласса *CDocument*. Их можно сделать открытыми, но, в целях лучшей защищенности, лучше их создать защищенными (*protected*), а доступ к ним организовать через специальные функции-члены. Объявим в классе *CTextDoc* заголовочного файла *TextDoc.h*, строку-объект *StringData* класса *CString*, в которой будем хранить введенные с клавиатуры символы:

```
class CTextDoc : public CDocument {
protected:
        CTextDoc();
        DECLARE_DYNCREATE(CTextDoc)
➧       CString StringData;
};
```

Инициализацию *StringData* пустой строкой выполним в конструкторе объекта *документ*, расположенном в файле *TextDoc.cpp*:

```
CTextDoc::CTextDoc() {
// TODO: add one-time construction code here
➧       StringData = " "; // инициализация пустой строкой
}
```

#### b) Чтение символов с клавиатуры

При нажатии клавиши Windows посылает сообщение *WM_CHAR*. Это сообщение надо связать с методом *OnChar* объекта *вид*. Для этого, используя утилиту *Class View*, выполним следующие действия:



1) вызовем *контекстное меню* класса *CTextView* (нажмем правой кнопкой по имени класса в окне утилиты *Class View*) → выберем пункт *Properties*;

2) в появившемся окне свойств нажмем на кнопку *Messages;*

3) выберем в списке сообщений *WM_CHAR*, раскроем список действий в поле напротив названия сообщения и выберем *Add OnChar*. Мастер установки *ClassWizard* присвоил имя *0nChar* обработчику сообщения *WM_CHAR* и включил в объект вида код для вызова этого метода:

```
void CTextView::OnChar(UINT nChar, UINT nRepCnt, UINT nFlags) {
        CView::OnChar(nChar, nRepCnt, nFlags);
}
```

В методе *0nChar* фактическое значение введенного символа передается в параметре *nChar*. Если пользователь не отпускает нажатую клавишу (что приводит к автоматической генерации нажатий), то количество нажатий будет передано в параметре *nRepCnt*. Введенный символ необходимо сохранить в объекте *StringData*, который принадлежит объекту *документ*. Поэтому вначале надо получить указатель *pDoc* на объект *документ* и затем добавить к строке *StringData* символ *nChar*:

```
void CTextView::OnChar(UINT nChar, UINT nRepCnt, UINT nFlags) {
// Получим указатель на объект документ
  ➧ CTextDoc* pDoc = GetDocument();
// Проверим полученный указатель
  ➧ ASSERT_VALID(pDoc);
// Сохраним и обновим строки символов
  ➧ pDoc->StringData += (char)nChar;
// Вызовем метод OnDraw для вывода строки
  ➧ Invalidate();
}
```

Макрос *ASSERT_VALID* проверяет, что полученный указатель *pDoc* действительно ссылается на объект *документ* (иначе ошибка).

#### c) Отображение введенного текста

Вывод текстовой строки выполним в методе *OnDraw* объекта *вид*. Для вызова метода *OnDraw* в программе используется функция *Invalidate*. Потребуем, чтобы текст выводился в центре клиентской области окна:

```
void CTextView::OnDraw(CDC* /*pDC*/) {
        CTextDoc* pDoc = GetDocument();
        ASSERT_VALID(pDoc);
        if (!pDoc) return;
        CDC* pDC = GetDC();
// Введем переменную rect для хранения размеров клиентской области окна
  ➧ CRect rect;
// Получим размеры клиентской области окна
  ➧ GetWindowRect(&rect);
// Зададим координаты x,y - центра клиентской области окна
  ➧ int x=rect.Width()/2;  int y=rect.Height()/2;
// Определим размеры выводимой строки в пикселях
  ➧ CSize size = pDC->GetTextExtent(pDoc->StringData);
// Сдвинем координаты x на половину длины выводимой строки влево
  ➧ x -= size.cx/2;
// Сдвинем координаты y на половину высоты выводимой строки вверх
  ➧ y -= size.cy/2;
// Выведем строку в центр клиентской области
  ➧ pDC->TextOut(x,y,pDoc->StringData);
}
```

После компиляции программа выводит набираемый текст в середину клиентской области окна.

### 3.4.4. Вывод битовых образов в окно программы

#### a) Создание битового образа

В Windows-приложениях графические объекты, которые можно нарисовать программно, чаще отображаются с помощью готовых битовых образов. Например, кнопки в нажатом и отпущенном состоянии, каркасы для целых окон и т. д. Поэтому битовые образы являются важной частью Windows. В MFC-приложении битовые образы описываются классом *CBitmap*, а для их создания обычно используют либо *ресурсный редактор*, либо готовые *bmp* файлы, созданные при помощи графических пакетов, которые импортируют в ресурсы. Битовые образы являются такими же ресурсами, как иконка или диалог, и используются чаще, чем все остальные ресурсы. Это объясняется наличием для них чрезвычайно мощной Windows-поддержки.



Битовые образы можно хранить и в отдельном файле с расширением *bmp* (единственный растровый формат, который напрямую поддерживается Windows). Однако необходимо помнить, что область ресурсов с битовыми образами в *exe*-файле может занимать большой размер. Но это не критично, так как ресурсы автоматически не загружаются в память.

#### b) Вывод битового образа на экран

Когда битовый образ помещен в ресурсы, его можно выводить на экран. Сначала необходимо создать объект типа *CBitmap*, загрузить в него битовый образ из ресурсов (или из *bmp* файла) и затем вывести в клиентскую область окна. Чаще всего в MFC-приложении для хранения изображений, которые затем копируются в заданное устройство вывода, применяется *контекст памяти*. Этот контекст создается совместимым с тем устройством или окном, на которое предполагается копировать информацию, т. е. он является переходником между программой и драйвером устройства.

Алгоритм работы вывода растрового образа с контекстом памяти состоит из нескольких шагов:

1. В программе объявим два контекста устройств. Первый контекст связан с текущим окном, а второй не инициализирован и предназначен для области памяти, в которой будет храниться растровое изображение.
2. С помощью функции *CreateCompatibleDC* контекст памяти объявим совместимым с текущим контекстом окна.
3. Выберем изображение *bitmap* как текущее для контекста памяти.
4. Копируем изображение из контекста памяти в текущий контекст устройства.
5. Удаление совместимого контекста устройства (*CompatibleDC*).
6. Удаляем объект *bitmap* из контекста памяти.
7. Освобождаем текущий контекст устройства (*DC*).

Именно этот алгоритм чаще всего используется в приложениях для вывода растровых изображений, поэтому разберем его более детально на следующем примере. Создадим новое Windows-приложение *Bitmap_test* и в нем организуем вывод растрового образа в клиентскую область окна. В

качестве *bmp* картинки используем образ планеты *Земля*, взятой из сайта http://www.systemplanet.narod.ru/zemlia.html.

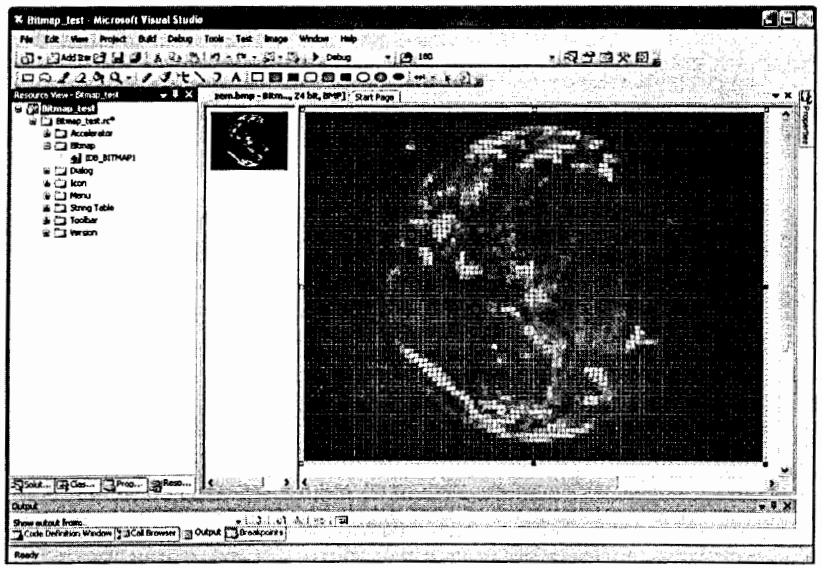

Рис.15. Загрузка растрового образа Земля в ресурсы проекта

Поместим файл с изображением Земли *zem.bmp* в ресурсы проекта *Bitmap_test*. Для этого в *контекстном меню* выбеем пункт *Меню* → *Project* → *Add Resource*. В открывшемся диалоговом окне выберем тип ресурса → *Bitmap* и опцию → *Import* и загрузим в ресурсы проекта файл *zem.bmp*. Мастер *Application Wizard* по умолчанию присвоил ему идентификатор *IDB_BITMAP1*. На рис.15 показан вид *Bitmap* ресурса с загруженным в него *bmp*-файлом. Этот рисунок содержит черный фон вокруг *Земли*, который при необходимости можно скорректировать прямо в ресурсе (закрасить кистью цвета фона окна) или можно выполнить корректировку программным методом (для этого необходимо создать регион отсечения типа эллипс и обрезать фон, окружающий рисунок).

Для вывода *bmp*-картинки *Земля* создадим в классе *CBitmap_testView* функцию *Bitmap_pict*, типа *void*, и в ней, используя рассмотренный выше алгоритм с контекстом памяти, организуем вывод из ресурсов растрового



образа в клиентскую область окна. Затем в этой функции создим объект контекста устройства в который будет осуществляться вывод растрового объекта *CClientDC dc(this)*. Для этого необходимо создать объект *bm* типа *CBitmap* и с помощью функции *LoadBitmap* загрузить в него битовый образ *Земли* из ресурсов проекта (рис.15), передав ему идентификатор *IDB_BITMAP1*.

На следующем шаге программы объявим контекст памяти *dcMem* для хранения растрового образа и с помощью функции *CreateCompatibleDC* сделаем его совместимым с текущим контекстом окна, в которое будем копировать изображение. Перед выводом на экран растровое изображение с помощью функции *SelectObject* должно быть выбрано в контекст памяти.

Для вывода изображения на экран используем функцию *BitBlt*, которая быстро копирует его из исходного контекста памяти в контекст, связанный с вызывающим функцию объектом. Прототип этой функции следующий:

```
BOOL CDC::BitBlt(int x, int y, int Width, int Height, CDC *pSourceDC, int SourceX, int SourceY,DWORD RasterOpCode);
```

Первые два параметра в этой функции задают координаты начальной точки вывода изображения, следующие два параметра – размеры выводимого изображения. Параметр *pSourceDC* является указателем на исходный контекст устройства. Координаты *SourceX* и *SourceY* задают левый верхний угол изображения (обычно равны 0). Последний параметр задает код операции, которая будет проделана при передаче изображения из одного контекста в другой. Мы будем использовать только значение *SRCCOPY*, в этом случае изображение просто копируется.

### c) Корректировка растрового образа, регионы отсечения

После компиляции программа выведет рисунок *Земля* с окружающим ее черным фоном. Для устранения этого эффекта выполним корректировку изображения с помощью *региона отсечения*. Регионы отсечения в MFC-программах используются для создания различных форм окон и областей рисования. Именно с помощью региона мы сможем ограничить область вывода объекта *Земля*, придавая ей нужную круглую форму, т. е. создать регион отсечения типа эллипс и обрезать фон, окружающий рисунок.

Для того чтобы в программе использовать регион, его вначале нужно создать. Так как регионы могут быть разной формы, то для их создания используются следующие функции:

1. *CreateRectRgn(int x1, int y1, int x2, int y2)* создает прямоугольный регион по заданным координатам.
2. *CreateRectRgnIndirect(LPCRECT lpRect)* создает прямоугольный регион, используя структуру *CRect*.
3. *CreateEllipticRgn(int x1, int y1, int x2, int y2)* создает эллипсоидный регион по заданным координатам.
4. *CreateEllipticRgnIndirect(LPCRECT lpRect)* создает эллипсоидный регион, используя структуру *CRect*.
5. *CreatePolygonRgn(LPPOINT lpPoints, int nCount, int nMode)* создает регион из массива точек.
6. *CreatePolyPolygonRgn(LPPOINT lpPoints, LPINT lpPolyCounts, int nCount, int nPolyFillMode)* создает регион из набора регионов.
7. *CreateRoundRectRgn(int x1, int y1, int x2, int y2, int x3, int y3)* создает регион с закругленными краями.

В нашей программе используем функцию *CreateEllipticRgnIndirect*, в которую в качестве параметров передадим параметры структуры *CRect*. Для этого в программе сначала надо получить размеры *bmp*-образа:

```
GetObject(sizeof(BITMAP), &bmp);
```

Создать структуру *rect*, описывающую квадрат вокруг объекта *Земля*:

```
CRect rect;
rect.left= x1;
rect.top = y1;
rect.right = x1+bmp.bmWidth;
rect.bottom=y1+bmp.bmHeight;
```

Здесь *x1*, *y1* – координаты вывода растрового образа. Затем надо создать регион отсечения и связать его с контекстом устройства, в который организуется вывод образа:

```
CRgn rgn;
rgn.CreateEllipticRgnIndirect(&rect);
```



```
SelectObject(&rgn);
```

Используя функцию *BitBlt*, копируем *bmp*-изображение в текущий контекст устройства по заданным параметрам вывода.

Окончательный код программы вывода растрового образа *Земля* в проекте *Bitmap_test* выглядит следующим образом:

```
void CBitmap_testView::Bitmap_pict() {
//1. Создадим в стеке объект контекста, в который будет осуществляться вывод
    CClientDC dc(this);
// 2. Создадим объект bmp класса BITMAP
    BITMAP bmp;
// 3. Создадим объект bm класса CBitmap
    CBitmap bm;
// 4. Загрузим в объект bm изображение из ресурсов, передав функции LoadBitmap
// идентификатор IDB_BITMAP1
    bm.LoadBitmap(IDB_BITMAP1);
// 5. Получим размеры загруженного растрового образа
    bm.GetObject(sizeof(BITMAP), &bmp);
// 6. Создадим контекст устройства памяти для хранения изображения
    CDC dcMem;
// 7. Объявим его совместимым с текущим контекстом устройства dc
    dcMem.CreateCompatibleDC(&dc);
// 8. Загрузим в него функцией SelectObject растровое изображение
    dcMem.SelectObject(bm);
// 9. Копируем это изображение на экран функцией BitBlt,
    dc.BitBlt(100,100,400,400,&dcMem,0,0,SRCCOPY);
// Корректировка изображения
// 10. Создадим структуру rect1, описывающую bmp-объект Земля
    CRect rect1;
    rect1.right = 300+bmp.bmWidth;
    rect1.top = 100;
    rect1.bottom=100+bmp.bmHeight;
    rect1.left= 300;
// 11. Создадим регион отсечения с параметрами структуры rect1
    CRgn rgn1;
    rgn1.CreateEllipticRgnIndirect(&rect1);
// 12. Свяжем его с контекстом, в который организуется вывод
    dc.SelectObject(&rgn1);
// 13. Копируем образ на экран:
```

```
➡ dc.BitBlt(300,100,600,600,&dcMem,0,0,SRCCOPY);
// 14. Создадим структуру rect2, описывающую квадрат вокруг объекта Земля
➡ CRect rect2;
➡ rect2.left = 500+16;
➡ rect2.top = 100+5;
➡ rect2.right = 500+bmp.bmWidth - 16;
➡ rect2.bottom = 100 +bmp.bmHeight - 5;
// 15. Создадим регион отсечения с параметрами структуры rect2
➡ CRgn rgn2;
// 16. Свяжем его с контекстом, в который организуется вывод
➡ rgn2.CreateEllipticRgnIndirect(&rect2);
➡ dc.SelectObject(&rgn2);
// 17. Копируем скорректированный образ Земли на экран
➡ dc.BitBlt(500,100,800,400,&dcMem,0,0,SRCCOPY);
// 18. Удалим контекст памяти
➡ dcMem.DeleteDC();
// 19. Удалим bm-объект
➡ bm.DeleteObject();
// 20. Удалим регионы отсечения
➡ dc.SelectClipRgn(NULL);
}
```

В данной программе мы выводим растровый объект *Земля* в трех различных режимах: вывод *bmp* растрового объекта (левая картинка, рис.16); вывод объекта в регион отсечения *эллипс* (центральная картинка, рис.16); вывод объекта в регион отсечения *круг* (правая картинка, рис.16), который полностью корректирует копируемое изображение.

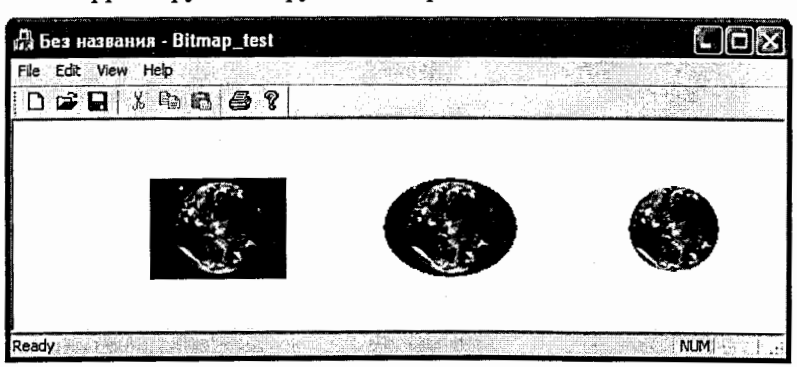


Рис.16. Вывод растрового изображения объекта *Земля*



## 3.5. Диалоговые окна и управляющие элементы в MFC-приложении

### 3.5.1. Типы диалоговых окон в Windows-приложении

В Windows-приложениях, кроме главного окна со строкой меню и специфическим для приложения содержимым, для взаимодействия между пользователем и программой применяются *диалоговые окна*. Обычно они используются для изменения настроек приложения и ввода информации. Например, практически все окна настроек приложения *Microsoft Word* являются диалоговыми. Диалоговое окно, как правило, содержит набор элементов управления, которые по типу являются *дочерними окнами*. С их помощью пользователь производит определенные управляющие воздействия на режим работы программы. В MFC в классе *CDialog* объявлен набор функций для создания отображения и управления содержимым диалогового окна, а его внешний вид разрабатывается с помощью *редактора ресурсов*.

В Windows используется четыре типа диалоговых окон:

1. *Модальные*. При отображении модального диалогового окна его *окно-владелец* приостанавливает свою работу. Пользователь сможет продолжить работу с ним только после завершения работы с модальным окном.
2. *Системные модальные*. При его активации приостанавливают работу все открытые приложения Windows.
3. *Немодальные*. При активации немодального диалогового окна работа его окна-владелеца не прерывается. Оно продолжает работать в обычном режиме. Эти окна удобны для непрерывного отображения важной для пользователя информации.
4. *Информационные диалоговые окна*. Это специальные модальные диалоговые окна, в которых выводится короткое сообщение для пользователя, заголовок и некоторая комбинация стандартных кнопок и пиктограмм. Эти окна предназначены для вывода коротких текстовых сообщений и запроса у пользователя ответа из нескольких стандартных вариантов (*Да*, *Нет*, *Отмена*, *OK*). Например, информационные окна часто применяются для уведомления пользователя об ошибках программы и для запроса варианта реакции на ошибку: повторение или отмена операции.

Взаимодействие между диалоговым окном и пользователем осуществляется с помощью *элементов управления*. Это особый тип окон для ввода или вывода информации. Элемент управления принадлежит окну-владельцу, в данном случае – диалоговому окну. Все версии Windows поддерживают некоторый набор стандартных элементов управления, к которым относятся: кнопки, контрольные переключатели, селекторные кнопки, списки, поля ввода, комбинированные списки, полосы прокрутки и статические элементы. Элементы управления способны как генерировать сообщения в ответ на действия пользователя, так и получать их от приложения. В последнем случае сообщения являются фактически командами, на которые элемент управления должен отреагировать. В MFC содержатся классы всех стандартных элементов управления. Эти классы порождаются от класса *CWnd*, описывают элементы, управления и содержат функции для работы с ними. Таким образом, элементы управления обладают характеристиками окна.

### 3.5.2.Создание модального диалогового окна

Большинство Windows-приложений при создании диалоговых окон пользуются ресурсами *шаблонов диалоговых окон*. Шаблон диалогового окна задает стиль, местоположение и размер окна и всех элементов управления внутри него. Он создается с помощью *редактора ресурсов* и является частью файла ресурсов, который входит в проект Windows-приложения.

Диалоговые окна редактируются пользователем визуально в редакторе ресурсов и вместе с элементами управления представляют собой один ресурс со своим идентификатором. Кроме того, каждый элемент управления имеет собственный идентификатор. Обычно эти идентификаторы имеют префикс в соответствии с названием данного элемента управления, хотя при желании можно использовать любые идентификаторы.

#### а) Создание диалогового окна с помощью мастера *AppWizard*

Рассмотрим процедуру создания диалогового окна в MFC-приложении. Создадим при помощи мастера установки *AppWizard* новый однодокументный проект приложения под названием *Example_dialog*. Затем в меню выберем



пункт *Project → Add Resource*. В открывшемся диалоговом окне *Add Resource* (рис.17) выберем *тип ресурса → Dialog* и укажем его *статус → New*.

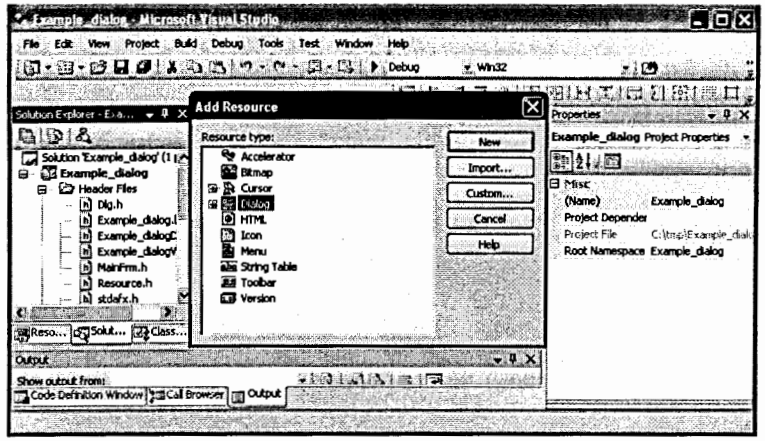


Рис.17. Создание диалогового окна с помощью редактора ресурсов

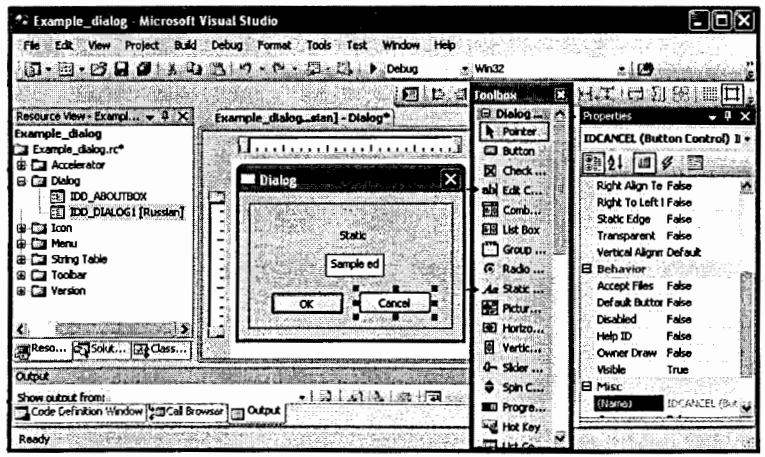


Рис.18. Редактирование диалогового окна с помощью редактора ресурсов

После этого этапа в нашем проекте в файле ресурсов появилось диалоговое окно (рис.18), которому редактор присвоил идентификатор *IDD_DIALOG1* и создал две кнопки: *OK* и *Cancel*. Для ввода в программу пользовательских данных нам потребуется добавить в диалоговое окно *текстовое поле*. Чтобы добавить новый элемент в диалоговое окно, нужно

мышкой «перетащить» требуемый элемент из инструментов *Tollbox* (рис.18). Перетащим текстовое поле (четвертый элемент сверху в редакторе) и *текстовую строку* (восьмой элемент сверху) в наше диалоговое окно. Можно посмотреть, какой идентификатор редактор по умолчанию присвоил каждому элементу диалогового окна. При необходимости можно изменить идентификатор или его название. Для этого щелчком правой клавиши мыши на выбранном элементе открываем окно с его свойствами (рис.19) и изменяем либо идентификатор (*IDC_EDIT*), либо текстовую строку. В нашем диалоговом окне мы меняем название элемента с идентификатором *IDC_STATIC* на «*Введенные данные*».

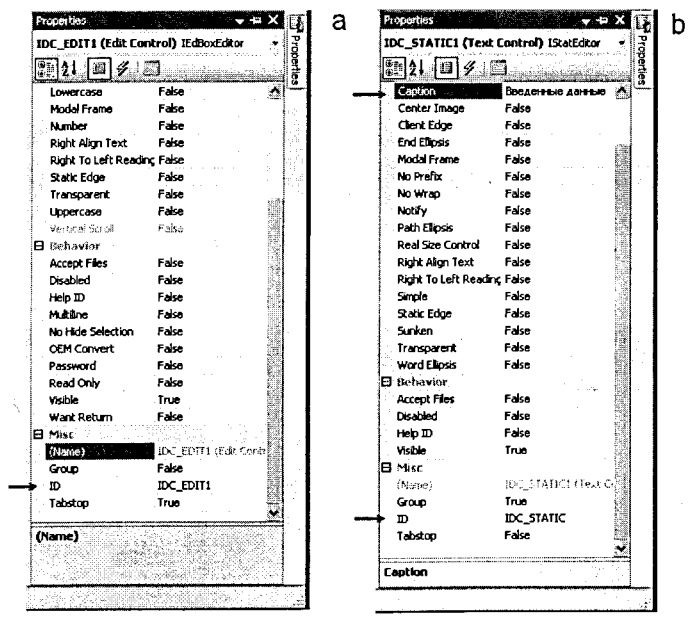


Рис.19. Параметры элементов диалогового окна: a) свойства элемента *текстовое поле*; b) свойства элемента *текстовая строка*

### b) Связывание элементов диалогового окна с переменными и методами обработки

Мы создали новое диалоговое окно, добавили в него два элемента и поменяли параметры одного из них. Это окно находится в файле ресурсов



*Example_dialog.rc*, а чтобы его использовать, необходимо включить его в программу и связать элементы диалогового окна с методами и переменными приложения. Для этого необходимо создать новый объект, связанный с ресурсом диалогового окна класса *CDialog*.

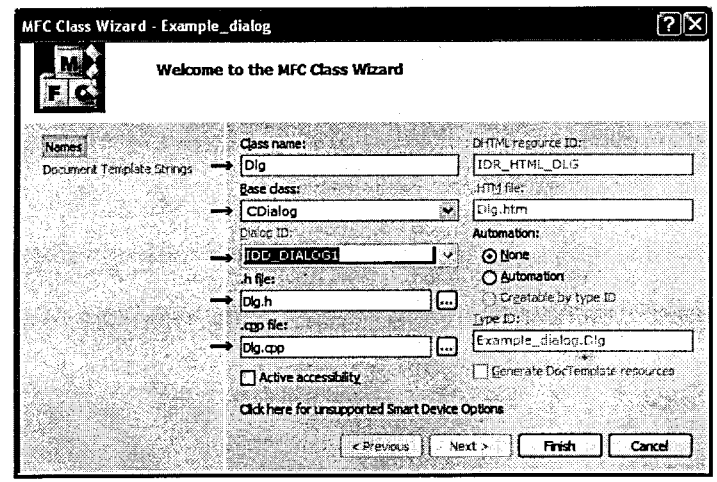


Рис.20. Создание нового класса *Dlg* в приложении *Example_dialog*

Для создания нового класса в приложении *Example_dialog* щелчком левой клавиши мыши на диалоговом окне вызовем мастера установки *AppWizard*. В открывшемся окне (рис.20), в поле *Class name*, зададим имя нашего класса *Dlg*. В поле *Base class* из списка выберем класс *CDialog*, который в MFC является базовым классом для диалоговых окон. Мастер *AppWizard* создал начальный базовый код этого класса (файлы *Dlg.cpp* и *Dlg.h*) и добавил их к проекту *Example_dialog*.

Затем осуществим процедуру связывания элементов диалогового окна с переменными и методами. Свяжем кнопку *OK* с методом обработки. Для этого в диалоговом окне щелчком правой клавиши мыши на этом элементе выберем опцию → *Add Event Handler* и в открывшемся окне *Event Handler Wizard* в поле *Message type* (рис.21) свяжем его с обработчиком сообщений → *BN_CLICKED*.

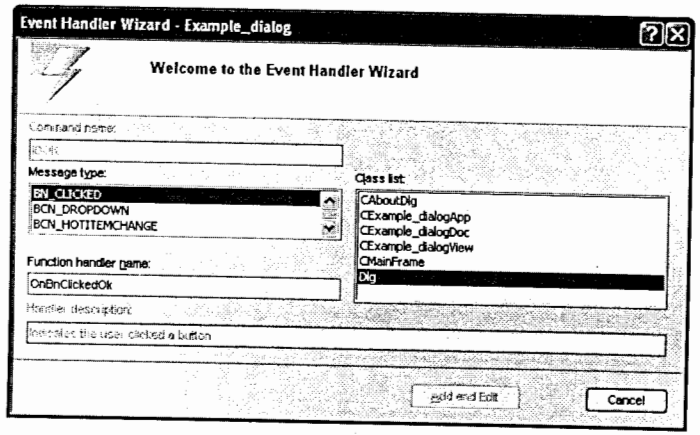


Рис.21. Связывание кнопки *OK* с методом обработки *BN_CLICKED*

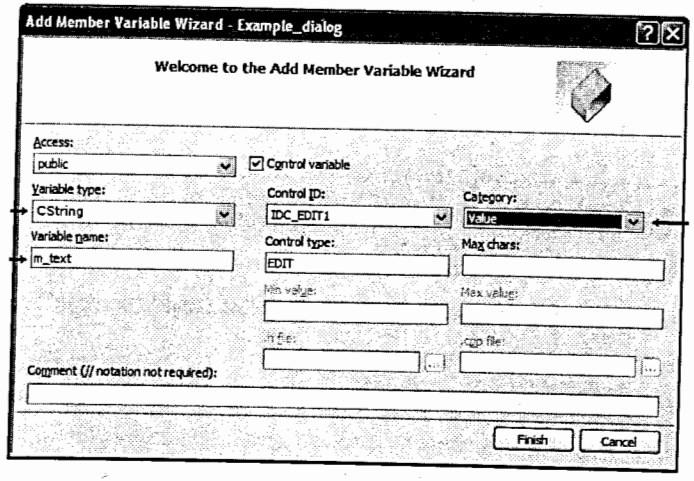


Рис.22. Создание элемента *текстовое поле* с переменной *m_text*

Свяжем элемент текстовое поле с переменной, в которую будет заноситься набранная в этом поле информация. В отличие от предыдущей процедуры, при связывании элемента текстовое поле выберем опцию → *Add Variable*. Затем в открывшемся окне *Add Member Variable Wizard* в поле *Category* (рис.22) зададим вид переменной → *Value*, в поле *Variable type* – ее тип → *CString*, а в поле *Variable name* – ее имя → *m_text*. После этой процедуры мы сможем передавать набранную информацию из текстового поля в строковую переменную *m_text*.



### c) Вызов диалогового окна в программе

Для вызова диалогового окна в программе приложения создадим, с помощью редактора ресурсов в строке меню главного окна (рис.23), новую команду *Dialog*. Для этого в окне редактора ресурсов (рис.23) откроем ресурс *Menu* и в окно *Type Here* занесем имя команды → *Dialog*, а в свойствах команды *Popup* зададим → *FALSE* (панель *Menu Editor*, рис.23). Редактор ресурсов по умолчанию присвоил этой команде идентификатор *ID_DIALOG*.

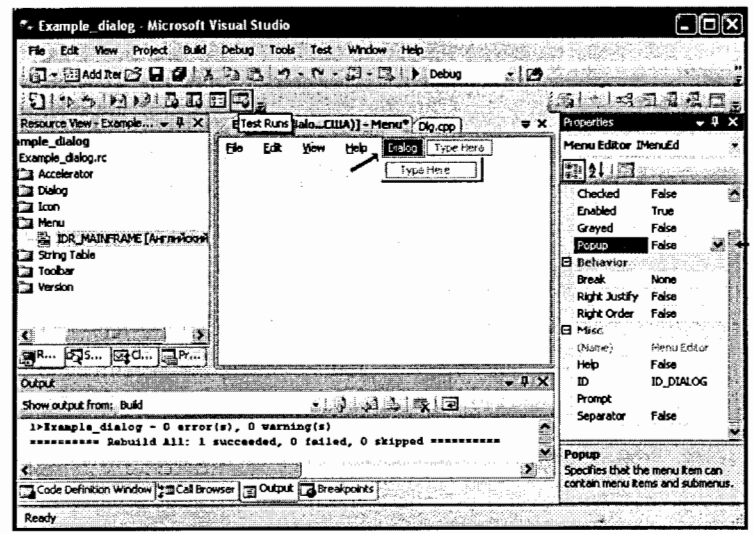


Рис.23. Создание новой команды меню *Dialog*

Диалоговое окно в нашем проекте должно вызваться при выборе в меню команды *Dialog*. Для этого в окне *MFC Class View* (рис.24) в классе *CExample_dialogView* свяжем идентификатор команды *ID_DIALOG* с методом обработки сообщения *COMMAND*. Мастер установки *AppWizard* создал начальный базовый код этой команды и добавил его в файл *Example_dialog.cpp*:

```
void CExample_dialogView::OnDialog()
{
➡       // TODO: Код обработки диалогового окна
}
```

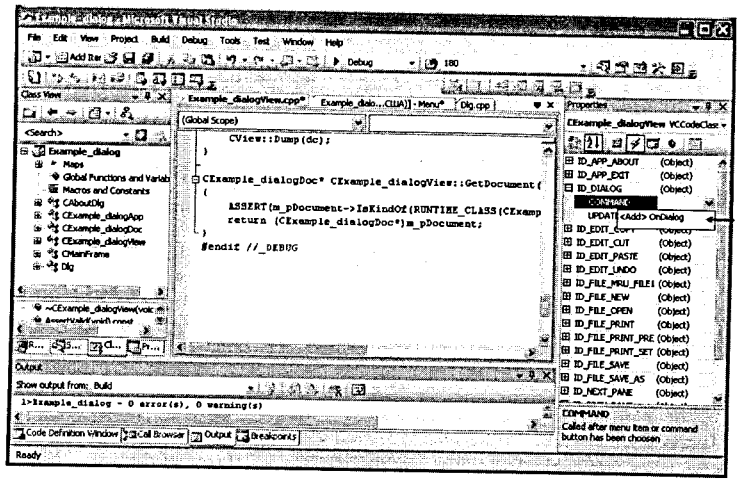

Рис.24. Связывание команды *Dialog* с методом обработки *COMMAND*

После разработки шаблона модального диалогового окна и объявления его подкласса для применения его в программе необходимо создать объект подкласса *CDialog* и вызвать у него функцию-член *DoModal*. Функция *DoModal* вернет управление программе только после закрытия диалогового окна, где в качестве возвращаемого значение будет передан параметр функции *EndDialog*. Приложение проверяет возвращенное значение и выполняет требуемые действия только в случае получения параметра *IDOK*. При другом возвращенном значении (обычно *IDCANCEL*) информация, введенная в диалоговом окне, игнорируется.

Конструктору класса *CDialog* в качестве параметров передается идентификатор ресурса шаблона и указатель на окно-владельца диалогового окна. В MFC обычно применяют конструктор, который можно использовать вообще без параметров. Такой конструктор позволяет создать и вызвать диалоговое окно всего двумя операторами:

```
CDialog dlg;
if ( dlg.DoModal() == IDOK ) // Пользователь нажал кнопку OK
{
// Код обработки диалогового окна
}
```



### d) Отображение информации из диалогового окна

Диалоговое окно в Windows-приложении предоставляет пользователю для выбора некоторый набор параметров, считывает введенные данные и делает их доступными приложению, создавшему окно. Удобным способом для хранения вводимых данных является использование переменных-членов класса диалогового окна. Приложение, пользующееся диалоговым окном, может изменять эти переменные-члены для инициализации окна или для получения данных после его закрытия.

Так как данные в MFC-программах хранятся в объекте *документ* в виде переменных-членов класса *CDocument*, то введенные в текстовом поле диалогового окна параметры можно получить через присвоение значению объекта *string* (которое хранится в объекте *документ*) значения переменной-члена диалогового окна. Для этого в объекте *документ* в классе *CExample_dialogDoc* объявим закрытую переменную *string* класса *CString*:

```
class CTextDoc : public CDocument {
        CExample_dialogDoc : public CDocument {
protected: // create from serialization only
        CExample_dialogDoc();
        DECLARE_DYNCREATE(CExample_dialogDoc)
        CString string;
}
```

В конструкторе присвоим этой переменной «пустое» значение:

```
CExample_dialogDoc::CExample_dialogDoc() {
        // TODO: add one-time construction code here
        string = " ";
}
```

Так как вывод строки текстового поля диалогового окна выполняется после нажатия кнопки *OK*, то вначале необходимо получить указатель *pDoc* на документ *CExample_dialogDoc* pDoc = GetDocument* и затем присвоить значение переменной *m_text* объекту *string*, т. е.:

```
void CExample_dialogView::OnDialog() {
        CExample_dialogDoc* pDoc = GetDocument();
        ASSERT_VALID(pDoc);
```

```
    Dlg dlg;        // Объявим объект dlg класса Dlg
    if ( dlg.DoModal() == IDOK ) {
    pDoc->string = dlg.m_text;          // Передадим значение m_text объекту string
    }
    Invalidate();  // Вызовем метод OnDraw для вывода параметра на экран
}
```

Для вывода объекта *string* на экран используем метод *OnDraw*:

```
void CExample_dialogView::OnDraw(CDC* /*pDC*/) {
    CExample_dialogDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    if (!pDoc)
    return;
    CDC* pDC = GetDC();
    pDC->TextOutA(10,10,pDoc->string);
}
```

После запуска приложения *Example_dialog* на экран выводится строка, набранная в текстовом поле диалогового окна (рис.25). В конкретных приложениях это может быть параметр, который необходимо ввести в программу.

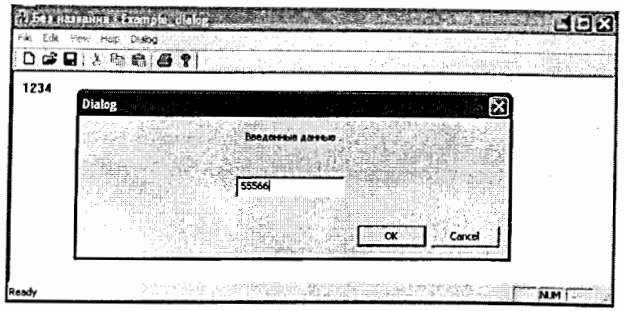


Рис.25. Создание диалогового окна в приложении *Example_dialog*

### 3.5.3.Создание элементов управления в MFC-приложении

Для диалога между пользователем и приложением кроме диалоговых окон применяют окна специального типа – *элементы управления*. Это могут быть кнопки, статические органы, списки, редакторы, полосы прокрутки и т. д. В MFC определен набор стандартных элементов управления, отличительной особенностью которых является то, что для них уже описаны классы



окон и вид обрабатываемых сообщений. При необходимости в приложении можно создавать собственные элементы управления, которые можно наследовать от стандартных оконных классов или разрабатывать «с нуля», как обычные окна, используя функцию *CreateWindow*. Для этого в программе в явном виде нужно указать параметры функции *CreateWindow*, имя оконного класса и стиль элемента управления. Класс элемента управления и стиль задаются в файле ресурсов приложения (например, стиль разновидность кнопки – нажимаемая, или с зависимой фиксацией). С помощью элементов управления в программе выполняются простые управляющие действия. Родительское окно получает сообщения, в которых содержится идентификатор *окна-отправителя* и требуемая команда.


Рис.26. Вид растровых объектов в проекте *Controls_Dialog*: a) растровая картинка фона; b) растровые объекты движения

Создание элементов управления в Windows-приложении рассмотрим на примере программы по имитации движения растровых объектов. В этом приложении мы будем выводить движущиеся растровые объекты (паровоз с различным числом вагонов, рис.26-b) на фоне растровой картины (рис.26-a). Выбранное количество вагонов в составе, направление движения и количество движущихся поездов в программу будем вводить при помощи стандарт-

ных управляющих элементов: скорость перемещения поездов будем задавать при помощи управляющего элемента *ползунок*; выбор направления движения растровых объектов – элементами *переключатель*; количество вагонов в составе – элементами *флажок*; начало движения – элементом *кнопка*.

### a) Создание стандартных управляющих элементов

С помощью мастера *AppWizard* создадим новый однодокументный проект с именем *Controls_Dialog*. Затем в этом проекте создадим шаблон диалогового окна и включим в него все необходимые управляющие элементы.

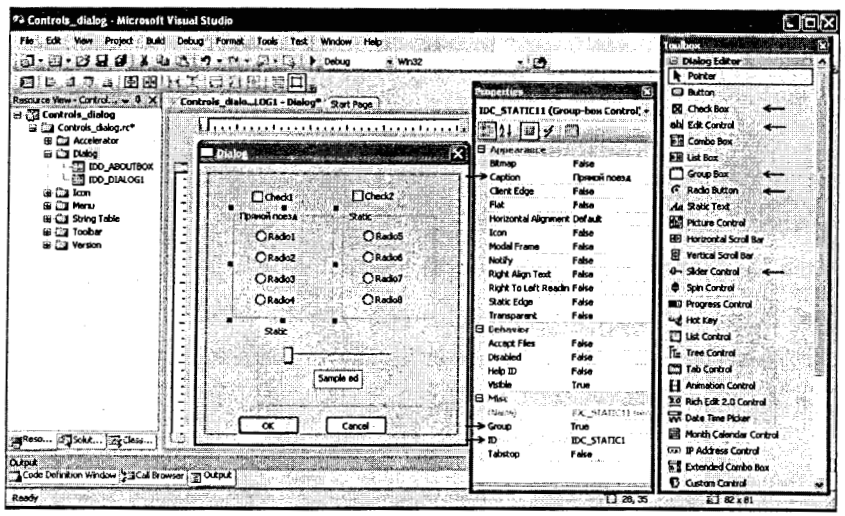


Рис.27. Создание шаблона диалогового окна с управляющими элементами

Для этого выполним следующие шаги (рис.27):

1. В редакторе ресурсов выберем *тип ресурса* → *Dialog* и *статус* → *New*.
2. В появившемся шаблоне диалогового окна создадим два групповых поля для включения в них переключателей типа *Radio Button* (из *Toolbox* перетащим два элемента *Group Box,* а для задания количества вагонов в составе перетащим в каждый из них по четыре переключателя типа *Radio Button*).
3. Для задания направления движения поездов из *Toolbox* добавим в шаблон диалогового окна два переключателя *Check Box* (*флажки*).
4. Создадим третье групповое поле и добавим в него управляющий элемент



*Slider Control* (для задания скорости перемещения поезда) и текстовое поле *Edit Control* (для ее визуализации).

Используя опцию *Properties*, в поле *Caption* (рис.27) каждому управляющему элемента поменяем название: *Check1* меняем на *прямой поезд*; *Check2 – встречный поезд*; *Static* верхнего левого группового поля меняем на *прямой поезд*; *Static* верхнего правого группового поля – на *встречный поезд*; *Radio1*и *Radio5* меняем на *Один вагон*; *Radio2*и *Radio6 – Два вагона*; *Radio3* и *Radio7 – Три вагона*; *Radio4* и *Radio8 – Четыре вагона*; *Static* нижнего группового поля меняем на *Скорость движения поезда*. На рис.28 показан внешний вид созданного шаблона диалогового окна с управляющими элементами для проекта *Controls_Dialog*.

Рис.28. Диалоговое окно в приложении *Controls_Dialog*

### b) Создание управляющего элемента типа ползунок

Управляющие элементы типа ползунок чаще всего применяются для ввода числовых величин или переменных параметров, принимающих какое-то значение из возможного диапазона. Ползунок содержит небольшой бегунок, перемещаемый пользователем вдоль шкалы значений. Когда пользователь перемещает бегунок мышью, программа вводит его значение по шкале от минимума (крайнее левое значение) до максимума (крайнее правое значение).

При создании диалогового окна редактор ресурсов присвоил ползунку идентификатор *IDC_SLIDER1*, а текстовому полю – *IDC_EDIT1*. С помощью *Class Wizard* в классе *CDlg* выберем опцию *Add Function* и в открывшемся

окне добавим два метода *OnInitDialog* и *OnHScroll*. Затем выберем опцию *Add Variable* и добавим переменную *m_slider* класса *CSliderCtrl*, связанную с идентификатором *IDC_SLIDER1*, и переменную *m_text1* класса *CString*, связанную с идентификатором текстового поля *IDC_EDIT1*.

При создании ползунка в приложении необходимо задать его интервал. Эта величина определяет возможные позиции бегунка от крайнего левого до крайнего правого положения. Зададим интервал от 1 до 20. Эти значения задаются методами *SetRangeMin* и *SetRangeMax* класса *CSliderCtrl*:

```
BOOL CDlg::OnInitDialog() {
        CDialog::OnInitDialog();
➧       m_slider.SetRangeMin(1, false);
➧       m_slider.SetRangeMax(20, false);
}
```

В этих функциях значение второго параметра *false* позволяет перерисовать бегунок после изменения интервала (*true* – отказаться от перерисовки).

Чтобы вывести в текстовом поле исходное положение бегунка (он находится в позиции 1), необходимо переменной *m_text*, связанной с содержимым текстового поля, присвоить значение 1, т. е.

```
BOOL CDlg::OnInitDialog() {
        CDialog::OnInitDialog();
        m_slider.SetRangeMin(1, false);
        m_slider.SetRangeMax(20, false);
➧       m_text1 = "1";
➧       UpdateData(false);
}
```

После того как ползунок создан и определено его начальное значение, необходимо описать код его перемещения. Когда пользователь перемещает бегунок, в диалоговое окно посылается сообщение *WM_HSCROLL* для метода *OnHScroll*(*UINT nSBCode, UINT nPos, CScrollBar* pScrollBar*). Параметрами этого метода являются: код операции прокрутки, новая позиция бегунка и указатель на управляющий элемент, от которого поступило сообщение. В нашем приложении от ползунка мы обрабатываем только сообщение *SB_THUMBPOSITION* (прокрутка в заданную позицию). Если в сообщении



содержится код прокрутки – это означает, что пользователь переместил бегунок и в текстовом поле необходимо вывести его новую позицию. Для этого в переменную текстового поля *m_text1* необходимо передать значение *nPos*, предварительно выполнив преобразование типа переменной (так как параметр *nPos* имеет целочисленный тип, а *m_text1* относится к классу *CString*), т. е.:

```
void CDlg::OnHScroll(UINT nSBCode, UINT nPos, CScrollBar* pScrollBar) {
➧       if(nSBCode == SB_THUMBPOSITION) {
➧           m_text1.Format("%ld", nPos);
➧           UpdateData(false);
        }
        else {
            CDialog::OnHScroll(nSBCode, nPos, pScrollBar);
        }
}
```

### c) Создание стандартного управляющего элемента типа кнопка

Для создания в проекте управляющего элемента типа *кнопка* в редакторе ресурсов выберем *тип ресурса* → *Menu* → *IDR_MAINFRAME* и с помощью *Toolbar* на панели *Menu* (рис.29) создадим стандартную нажимаемую кнопку.

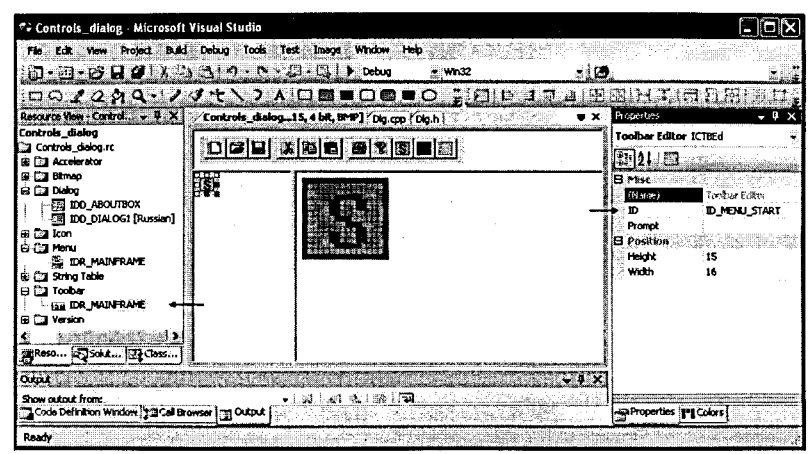


Рис.29. Создание управляющего элемента *кнопка*

### d) Создание класса диалогового окна и связывание управляющих элементов и переменных с методами обработки

Для создания класса диалогового окна (щелчком правой клавиши мыши на диалоговом окне) вызовем *AppWizard* и в открывшемся окне, в поле *Class name*, зададим имя нашего класса *CDlg*, а в поле *Base class – CDialog*, (базовый класс для диалоговых окон). Мастер *AppWizard* создал начальный базовый код этого класса (файлы *Dlg.cpp* и *Dlg.h*) и добавил его к проекту *Controls_Dialog*. После создания класса необходимо произвести процедуру связывания элементов диалогового окна с переменными и методами.

Свяжем флажок *Check1*, который задает прямое направление движения объекта, с методом обработки. Для этого в диалоговом окне выберем опцию *Add Event Handler* и свяжем этот элемент с обработчиком сообщений *BN_CLICKED* и здесь же, используя опцию *Add Variable*, добавим в созданный метод переменную *m_check1 = true* типа *bool*:

```
void CDlg::OnBnClickedCheck1() {
➧     m_check1 = true;
}
```

Аналогичную процедуру связывания выполним для элемента *Check2* (обратное направление движения поезда) и переключателей *Radio1 – Radio8* (количество вагонов в составе поезда), только в метод *OnBnClickedCheck2* добавим переменную *m_check2 = true*, типа *booll*; в *OnBnClickedRadio1 – FR1_d = true*; в *OnBnClickedRadio2 – FR2_d = true*; в *OnBnClickedRadio3 – FR3_d = true*; в *OnBnClickedRadio4 – FR4_d = true*; в *OnBnClickedRadio5 – FR1_r = true*; в *OnBnClickedRadio6 – FR2_r = true*; в *OnBnClickedRadio7 – FR3_r = true*; в *OnBnClickedRadio8 – FR4_r = true*.

Для вызова диалогового окна и кнопок управления на панели меню в программе приложения с помощью редактора ресурсов создадим в строке *Menu* главного окна (рис.30) новые команды: *Dialog*, *Start* и *Stop* и свяжем их в классе *CControls_dialogView* с обработчиками сообщений типа *COMMAND*:

```
void CControls_dialogView::OnMenuStart() {
➧     FLS = true;
➧     Invalidate();
}
void CControls_dialogView::OnMenuStop() {
➧     FLS = false;
➧     Invalidate();
}
void CControls_dialogView::OnMenuDialog() {
      // код обработки диалогового окна
}
```



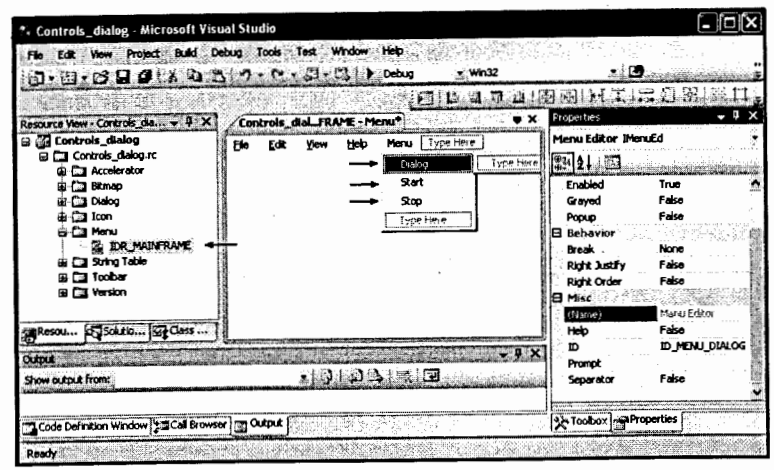


Рис.30. Создание на панели меню команд *Dialog*, *Start* и *Stop*

Затем для кнопок управления *cmapm* и *cmon* меняем идентификаторы на *ID_MENU_START* и *ID_MENU_STOP* соответственно.

Для управления параметрами программы в объекте вида создадим переменные *FFD*, *FFR*, *FL1_d*, *FL2_d*, *FL3_d*, *FL4_d*, *FL1_r*, *FL2_r*, *FL3_r*, *FL4_r* типа *bool*, а в методе *OnMenuDialog* передадим им соответствующие значения переменных из диалогового окна:

```
void CControls_dialogView::OnMenuDialog() {
      CDlg dlg;
      if ( dlg.DoModal() == IDOK ) {
      ➧ FFD = dlg.m_check1;  FFR = dlg.m_check2;
      ➧ FL1_d = dlg.FR1_d;   FL2_d = dlg.FR2_d;
      ➧ FL3_d = dlg.FR3_d;   FL4_d = dlg.FR4_d;
      ➧ FL1_r = dlg.FR1_r;   FL2_r = dlg.FR2_r;
```

```
    FL3_r = dlg.FR3_r;      FL4_r = dlg.FR4_r;
    Delay = atoi(dlg.m_text1);
    }
}
```

Таким образом, мы разработали *интерфейс пользователя*, при помощи которого будем изменять параметры функционирования программы.

### е) Создание и вывод растровых объектов на экран

В нашем проекте для вывода растровых объектов применим уже известный нам алгоритм с использованием контекста памяти (подпункт 3.4.4), добавив в него необходимые дополнения. Нам потребуется создать девять контекстов памяти, по числу выводимых объектов, и создать специальную кисть фона для восстановления (закраски фоном) локальной области картинки при перемещении растрового объекта (паровоза с вагонами) в следующую текущую позицию (имитация движения).

Алгоритм работы программы по выводу растровых объектов следующий:

1. Создадим контекст устройства *CDC*, связанный с текущим окном и девять контекстов памяти (*dcMem1_d*, *dcMem1_r*, *dcMem2_d*, *dcMem2_r*, *dcMem3_d*, *dcMem3_r*, *dcMem4_d*, *dcMem4_r*, *dcMem5*) для хранения растровых объектов изображения (паровозы с различным количеством вагонов).
2. Объявим эти контексты памяти совместимыми с текущим контекстом окна и загрузим в них из ресурсов проекта растровые объекты.
3. Копируем растровое изображение картинки фона в контекст устройства и создаем текущую кисть фона, которую будем использовать для закрашивания прежнего положения картинки фона при перемещении движущегося объекта.
4. В цикле копируем из контекстов памяти в текущее окно приложения выбранные, с помощью управляющих элементов диалогового окна растровые картинки (паровозы с вагонами). Цикл запускается от кнопки *пуск* и в него с помощью ползунка передаем скорость перемещения растровых объектов.
5. Удаляем совместимые контексты устройств и объекты.
6. Освобождаем текущий контекст устройства.



Полный код программы, описывающий данный алгоритм работы, приведен в листингах файлов *Controls_dialogView.cpp* и *Dlg.cpp*, а на рис.31 показан вид окна приложения с двумя движущимися составами, в котором параметры управления заданы при помощи диалогового окна (справа вверху).

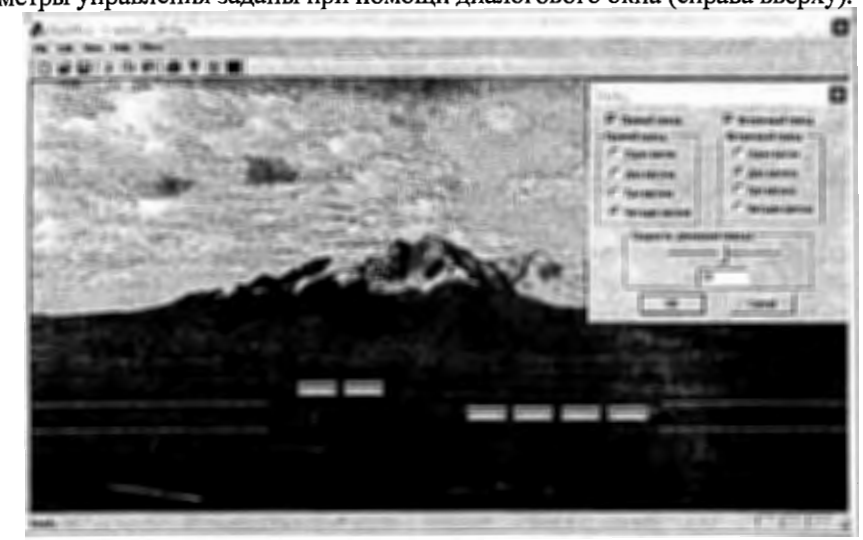

Рис.31. Вывод движущихся объектов в окно приложения *Controls_dialog*

```
// Controls_dialogView.cpp
#include "stdafx.h"
#include "Controls_dialog.h"
#include "Controls_dialogDoc.h"
#include "Controls_dialogView.h"
#include "Dlg.h"
#ifdef _DEBUG
#define new DEBUG_NEW
#endif
// CControls_dialogView
IMPLEMENT_DYNCREATE(CControls_dialogView, CView)
BEGIN_MESSAGE_MAP(CControls_dialogView, CView)
        ON_COMMAND(ID_FILE_PRINT, &CView::OnFilePrint)
        ON_COMMAND(ID_FILE_PRINT_DIRECT, &CView::OnFilePrint)
        ON_COMMAND(ID_FILE_PRINT_PREVIEW, &CView::OnFilePrintPreview)
        ON_COMMAND(ID_MENU_DIALOG, &CControls_dialogView::OnMenuDialog)
        ON_COMMAND(ID_MENU_START, &CControls_dialogView::OnMenuStart)
```

```
        ON_COMMAND(ID_MENU_STOP, &CControls_dialogView::OnMenuStop)
END_MESSAGE_MAP()
// CControls_dialogView construction/destruction
CControls_dialogView::CControls_dialogView() {
➧       FL1_d = FL2_d = FL3_d = FL4_d = FL1_r = FL2_r = FL3_r = FL4_r = FFD =false;
➧       FFR = FLS =false; Delay = 1;
}
CControls_dialogView::~CControls_dialogView() {
}
BOOL CControls_dialogView::PreCreateWindow(CREATESTRUCT& cs) {
        return CView::PreCreateWindow(cs);
}
// CControls_dialogView drawing
void CControls_dialogView::OnDraw(CDC* /*pDC*/) {
        CControls_dialogDoc* pDoc = GetDocument();
        ASSERT_VALID(pDoc);
        if (!pDoc)
        return;
➧       if(FLS) {
        ➧  Pict();
        }
}
// CControls_dialogView printing
BOOL CControls_dialogView::OnPreparePrinting(CPrintInfo* pInfo) {
        return DoPreparePrinting(pInfo);
}
void CControls_dialogView::OnBeginPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/) {
}
void CControls_dialogView::OnEndPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/) {
}
// CControls_dialogView diagnostics
#ifdef _DEBUG
void CControls_dialogView::AssertValid() const {
        CView::AssertValid();
}
void CControls_dialogView::Dump(CDumpContext& dc) const {
        CView::Dump(dc);
}
CControls_dialogDoc* CControls_dialogView::GetDocument() const // non-debug ver-
sion is inline {
```



```
        ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CControls_dialogDoc)));
        return (CControls_dialogDoc*)m_pDocument;
}
#endif //_DEBUG
// CControls_dialogView message handlers
void CControls_dialogView::OnMenuDialog() {
➧       CDlg dlg;
        if ( dlg.DoModal() == IDOK ) {
        ➧  FFD   = dlg.m_check1; FFR   = dlg.m_check2;
        ➧  FL1_d = dlg.FR1_d;    FL2_d = dlg.FR2_d;
        ➧  FL3_d = dlg.FR3_d;    FL4_d = dlg.FR4_d;
        ➧  FL1_r = dlg.FR1_r;    FL2_r = dlg.FR2_r;
        ➧  FL3_r = dlg.FR3_r;    FL4_r = dlg.FR4_r;
        ➧  Delay = atoi(dlg.m_text1);
        }
}
void CControls_dialogView::OnMenuStart() {
➧    FLS = true;
➧    Invalidate();
}
void CControls_dialogView::OnMenuStop() {
➧    FLS = false;
}
void CControls_dialogView::Pict() {
➧    CRect rect1,rect2;
// Получим контекст устройства, в который будет осуществляться вывод изображения.
➧    CClientDC *pDC = new CClientDC(this);
// Создадим объект bmp, класса BITMAP
➧    BITMAP bmp1,bmp2,bmp3,bmp4;
// Объявим контексты устройства памяти.
➧    CDC dcMem1_d,dcMem1_r,dcMem2_d,dcMem2_r,dcMem3_d;
➧    CDC  dcMem3_r,dcMem4_d,dcMem4_r,dcMem5;
// Создадим объекты cb, класса CBitmap
➧    CBitmap cb1_d,cb1_r,cb2_d,cb2_r,cb3_d,cb3_r,cb4_d,cb4_r,cb5;
// Загрузим в них объекты, передав функциии LoadBitmap идентификаторы из ресурсов.
➧    cb5.LoadBitmap(IDB_BITMAP1); cb4_d.LoadBitmap(IDB_BITMAP2);
➧    cb4_r.LoadBitmap(IDB_BITMAP3); cb3_d.LoadBitmap(IDB_BITMAP4);
➧    cb3_r.LoadBitmap(IDB_BITMAP5); cb2_d.LoadBitmap(IDB_BITMAP6);
➧    cb2_r.LoadBitmap(IDB_BITMAP7); cb1_d.LoadBitmap(IDB_BITMAP8);
➧    cb1_r.LoadBitmap(IDB_BITMAP9);
```

```
// Получим размеры загруженного растрового объекта cb
    if(FL4_d||FL4_r) {
        cb4_d.GetObject(sizeof(BITMAP), &bmp4);
    }
    if(FL3_d||FL3_r) {
        cb3_d.GetObject(sizeof(BITMAP), &bmp3);
    }
    if(FL2_d||FL2_r) {
        cb2_d.GetObject(sizeof(BITMAP), &bmp2);
    }
    if(FL1_d||FL1_r) {
        cb1_d.GetObject(sizeof(BITMAP), &bmp1);
    }
// Сделаем контексты памяти совместимыми с CDC
    dcMem1_d.CreateCompatibleDC(pDC); dcMem1_r.CreateCompatibleDC(pDC);
    dcMem2_d.CreateCompatibleDC(pDC); dcMem2_r.CreateCompatibleDC(pDC);
    dcMem3_d.CreateCompatibleDC(pDC); dcMem3_r.CreateCompatibleDC(pDC);
    dcMem4_d.CreateCompatibleDC(pDC);dcMem4_r.CreateCompatibleDC(pDC);
    dcMem5.CreateCompatibleDC(pDC);
// Поместим в контексты памяти растровые объекты функцией SelectObject
    dcMem1_d.SelectObject(cb1_d);  dcMem1_r.SelectObject(cb1_r);
    dcMem2_d.SelectObject(cb2_d);  dcMem2_r.SelectObject(cb2_r);
    dcMem3_d.SelectObject(cb3_d);  dcMem3_r.SelectObject(cb3_r);
    dcMem4_d.SelectObject(cb4_d);dcMem4_r.SelectObject(cb4_r);
    dcMem5.SelectObject(cb5);
// Выведем картинку cb5 (IDB_BITMAP1) - фон
    pDC->BitBlt(0,0,1400,600,&dcMem5,0,0,SRCCOPY);
// Создадим кисть фона
    brush.CreatePatternBrush(&cb5);
//  Выведем картинку IDB_BITMAP2 в цикле
    for(int i = 0; i<740;i++) {
        if(FL4_d||FL4_r) {
//  Зададим размеры области rect1
            rect1.left = -1+i;     rect1.top = 478;
            rect1.right = -1+bmp4.bmWidth+i;
            rect1.bottom=478+bmp4.bmHeight;
// Задаем размеры области rect2
            rect2.left = 738-i;   rect2.top = 449;
            rect2.right = 738-i+bmp4.bmWidth+2;
            rect2.bottom=449+bmp4.bmHeight;
```



```
        }
        if(FL3_d||FL3_r) {
// Зададим размеры области rect1
            rect1.left = -1+i;     rect1.top = 478;
            rect1.right = -1+bmp3.bmWidth+i;
            rect1.bottom=478+bmp3.bmHeight;
// Зададим размеры области rect2
            rect2.left = 738-i;   rect2.top = 449;
            rect2.right = 738-i+bmp3.bmWidth+2;
            rect2.bottom=449+bmp3.bmHeight;
    }
        if(FL2_d||FL2_r) {
// Зададим размеры области rect1
            rect1.left = -1+i; rect1.top = 478;
            rect1.right = -1+bmp2.bmWidth+i;
            rect1.bottom=478+bmp2.bmHeight;
// Задаем размеры области rect2
            rect2.left = 738-i; rect2.top = 449;
            rect2.right = 738-i+bmp2.bmWidth+2;
            rect2.bottom=449+bmp2.bmHeight;
    }
        if(FL1_d||FL1_r) {
// Зададим размеры области rect1
            rect1.left = -1+i;     rect1.top = 478;
            rect1.right = -1+bmp2.bmWidth+i;
            rect1.bottom=478+bmp2.bmHeight;
// Зададим размеры области rect2
            rect2.left = 738-i;   rect2.top = 449;
            rect2.right = 738-i+bmp2.bmWidth+2;
            rect2.bottom=449+bmp2.bmHeight;
    }
// Скопируем картинку на экран
    if(FFD&FL4_d) {
        pDC->FillRect(&rect1,&brush);
        pDC->BitBlt(0+i,478,1000,600,&dcMem4_d,0,0,SRCCOPY);
    }
    if(FFR&FL4_r){
        pDC->FillRect(&rect2,&brush);
        pDC->BitBlt(739-i,449,1000,600,&dcMem4_r,0,0,SRCCOPY);
    }
```

```
        if(FFD&FL3_d) {
            pDC->FillRect(&rect1,&brush);
            pDC->BitBlt(0+i,478,1000,600,&dcMem3_d,0,0,SRCCOPY);
        }
        if(FFR&FL3_r) {
            pDC->FillRect(&rect2,&brush);
            pDC->BitBlt(739-i,449,1000,600,&dcMem3_r,0,0,SRCCOPY);
        }
        if(FFD&FL2_d) {
            pDC->FillRect(&rect1,&brush);
            pDC->BitBlt(0+i,478,1000,600,&dcMem2_d,0,0,SRCCOPY);
        }
        if(FFR&FL2_r) {
            pDC->FillRect(&rect2,&brush);
            pDC->BitBlt(739-i,449,1000,600,&dcMem2_r,0,0,SRCCOPY);
        }
        if(FFD&FL1_d) {
            pDC->FillRect(&rect1,&brush);
            pDC->BitBlt(0+i,478,1000,600,&dcMem1_d,0,0,SRCCOPY);
        }
        if(FFR&FL1_r) {
            pDC->FillRect(&rect2,&brush);
            pDC->BitBlt(739-i,449,1000,600,&dcMem1_r,0,0,SRCCOPY);
        }
        Sleep(int(100/Delay));
    }
// Удалим контексты памяти
    dcMem1_d.DeleteDC(); dcMem2_d.DeleteDC(); dcMem3_d.DeleteDC();
    dcMem5_d.DeleteDC(); dcMem1_r.DeleteDC(); dcMem2_r.DeleteDC();
    dcMem3_r.DeleteDC();dcMem4_r.DeleteDC(); dcMem5.DeleteDC();
// Удалим cb объекты
    cb1_d.DeleteObject(); cb2_d.DeleteObject(); cb3_d.DeleteObject();
    cb4_d.DeleteObject(); cb1_r.DeleteObject();cb2_r.DeleteObject();
    cb3_r.DeleteObject();cb1_r.DeleteObject(); cb5.DeleteObject();
// Освободим текущий контекст устройства
    ReleaseDC(pDC);
}
```

*// Dlg.cpp*

```
#include "stdafx.h"
```



```
#include "Controls_dialog.h"
#include "Dlg.h"
// CDlg dialog
IMPLEMENT_DYNAMIC(CDlg, CDialog)
CDlg::CDlg(CWnd* pParent /*=NULL*/)
    : CDialog(CDlg::IDD, pParent)
    , FR1_d(false) , FR2_d(false) , FR3_d(false) , FR4_d(false) , FR1_r(false)
    , FR2_r(false), FR3_r(false), FR4_r(false), m_text1(_T(""))
{
    m_check1 = false; m_check2 = false;
}
CDlg::~CDlg() {
}
void CDlg::DoDataExchange(CDataExchange* pDX) {
    CDialog::DoDataExchange(pDX);
    DDX_Control(pDX, IDC_SLIDER1, m_slider);
    DDX_Text(pDX, IDC_EDIT1, m_text1);
}
BEGIN_MESSAGE_MAP(CDlg, CDialog)
    ON_BN_CLICKED(IDC_RADIO3, &CDlg::OnBnClickedRadio3)
    ON_BN_CLICKED(IDC_RADIO4, &CDlg::OnBnClickedRadio4)
    ON_BN_CLICKED(IDC_RADIO1, &CDlg::OnBnClickedRadio1)
    ON_BN_CLICKED(IDC_RADIO2, &CDlg::OnBnClickedRadio2)
    ON_BN_CLICKED(IDC_RADIO5, &CDlg::OnBnClickedRadio5)
    ON_BN_CLICKED(IDC_RADIO6, &CDlg::OnBnClickedRadio6)
    ON_BN_CLICKED(IDC_RADIO7, &CDlg::OnBnClickedRadio7)
    ON_BN_CLICKED(IDC_RADIO8, &CDlg::OnBnClickedRadio8)
    ON_BN_CLICKED(IDC_CHECK2, &CDlg::OnBnClickedCheck2)
    ON_BN_CLICKED(IDC_CHECK1, &CDlg::OnBnClickedCheck1)
    ON_WM_HSCROLL()
END_MESSAGE_MAP()
// CDlg message handlers
void CDlg::OnBnClickedRadio1() {
    FR1_d = true; FR2_d = false; FR3_d = false; FR4_d = false;
}
void CDlg::OnBnClickedRadio2() {
    FR1_d = false; FR2_d = true; FR3_d = false; FR4_d = false;
}
void CDlg::OnBnClickedRadio3() {
    FR1_d = false; FR2_d = false; FR3_d = true; FR4_d = false;
```

```
}
void CDlg::OnBnClickedRadio4() {
    FR4_d = true; FR1_r = false; FR2_r = false; FR3_r = false; FR4_r = false;
}
void CDlg::OnBnClickedRadio5() {
    FR1_r = true; FR2_r = false; FR3_r = false; FR4_r = false;
}
void CDlg::OnBnClickedRadio6() {
    FR1_r = false; FR2_r = true; FR3_r = false; FR4_r = false;
}
void CDlg::OnBnClickedRadio7() {
    FR1_r = false; FR2_r = false; FR3_r = true; FR4_r = false;
}
void CDlg::OnBnClickedRadio8() {
    FR1_r = false; FR2_r = false; FR3_r = false; FR4_r = true;
}
void CDlg::OnBnClickedCheck2() {
    m_check1 = true;
}
void CDlg::OnBnClickedCheck1() {
    m_check2 = true;
}
void CDlg::OnHScroll(UINT nSBCode, UINT nPos, CScrollBar* pScrollBar) {
    if(nSBCode == SB_THUMBPOSITION)  {
        m_text1.Format("%ld", nPos);
        UpdateData(false);
        }
        else {
        CDialog::OnHScroll(nSBCode, nPos, pScrollBar);
        }
}
BOOL CDlg::OnInitDialog() {
        CDialog::OnInitDialog();
    m_slider.SetRangeMin(1, false);
    m_slider.SetRangeMax(20, false);
    m_text1 = "1";
    UpdateData(false);
        return TRUE;
}
```



## 3.6. Ввод и обработка информации от мыши в MFC-приложении

### 3.6.1. Работа с мышью в MFC-приложении

В Windows клавиатура и мышь являются основными устройствами ввода информации в программу. Многие операции с мышью Windows выполняет автоматически, например, при активировании меню операционная система отслеживает выбор в нем пункта, а затем посылает программе сообщение *WM_COMMAND* с кодом выбранной команды. При нажатии клавиши мыши или при ее перемещении мыши Windows генерирует прерывание, которое обрабатывается драйверами устройств. Они помещают эту информацию в общую системную очередь, называемую *очередью необработанного ввода*. Специальный системный поток отслеживает содержимое очереди необработанного ввода и перемещает каждое обнаруженное сообщение в очередь сообщений соответствующего потока.

#### а) Типы и обработчики сообщений от мыши

В Windows с событиями от мыши связаны более 20 сообщений, которые можно разделить на сообщения, связанные с клиентской областью окна, и сообщения, не связанные клиентской областью. Они одинаковы по смыслу, но различаются положением указателя в момент возникновения события. События бывают следующими: нажатие или отпускание кнопки мыши; двойной щелчок кнопкой мыши; перемещение мыши. Сообщения, которые начинаются с имени *WM_LBUTTON*, относятся к левой кнопке мыши, *WM_MBUTTON* – к средней кнопке, а *WM_RBUTTON* – к правой кнопке. Имена функций-членов класса *CWnd* для обработки сообщений мыши приведены в табл.10.

Прототипы обработчиков сообщений от мыши одинаковы, например, *OnLButtonDown* имеет следующий вид: *afx_msg void OnMsgName(UINT nFlags, CPoint point)*. Параметр *point* содержит координаты указателя в момент возникновения события от мыши. Эти координаты задаются в физической системе координат, связанной с левым верхним углом клиентской области окна. Параметр *nFlags* содержит состояние кнопок мыши в момент генерации сообщения, которое можно извлечь из параметра *nFlags* с по-

мощью соответствующих масок: *MK_LBUTTON* (нажата левая кнопка мыши); *MK_MBUTTON* (нажата средняя кнопка мыши); *MK_RBUTTON* (нажата правая кнопка мыши).

Таблица 10. Сообщения мыши, связанные с клиентской областью окна

| Сообщение | Когда посылается |
|---|---|
| *WM_LBUTTONDOWN* | Нажата левая кнопка мыши |
| *WM_LBUTTONUP* | Левая кнопка мыши отпущена |
| *WM_LBUTTONDBLCLK* | Двойной щелчок левой кнопкой мыши |
| *WM_MBUTTONDOWN* | Нажата средняя кнопка мыши |
| *WM_MBUTTONUP* | Средняя кнопка мыши отпущена |
| *WM_MBUTTONDBLCLK* | Двойной щелчок средней кнопкой мыши |
| *WM_RBUTTONDOWN* | Нажата правая кнопка мыши |
| *WM_RBUTTONUP* | Правая кнопка мыши отпущена |
| *WM_RBUTTONDBLCLK* | Двойной щелчок правой кнопкой мыши |
| *WM_MOUSEMOVE* | Указатель мыши перемещается над клиентской областью окна |

Сообщения от мыши, связанные с неклиентской областью, аналогичны рассмотренным выше. Только в именах констант добавляются символы *NC*, например, вместо *WM_LBUTTONDOWN – WM_NCLBUTTONDOWN*, но эти сообщения обрабатываются в приложениях значительно реже.

### b) Режим захвата мыши

Многие приложения реагируют только на сообщения о нажатии кнопок мыши. Если в программе (например, при рисовании или при выделении объектов с помощью «резинового контура») необходимо обрабатывать нажатие и отпускание кнопки, то приложение должно использовать режим «захвата» мыши. Приложение (точнее, окно приложения) может «захватить» мышь при получении сообщения о нажатии кнопки. Тогда окно будет получать все сообщения мыши, независимо от того, где находится ее указатель. При получении сообщения об отпускании кнопки приложение может «освободить» мышь. Обычно вызовы этих функций располагаются в обработчиках нажатия и отпускания кнопки мыши, например:

```
if(nFlags&&MK_LBUTTON) { // код обработки сообщения }
```



## 3.6.2. Разработка MFC-приложения с обработкой сообщений мыши

С помощью *AppWizard* создадим новый проект Windows-приложения с именем *Painter*. В этом проекте мы будем рисовать различные графические объекты (типа эллипс, квадрат, линия и др.) по координатам, вводимым с помощью мыши. Интерфейс пользователя в программе будет состоять из диалогового меню, с помощью которого будет выбираться цвет и толщина пера для рисования графических объектов, стандартных управляющих элементов типа переключатель, кнопки и курсор мыши.

### a) Ввод информации от мыши

Для создания графических объектов с помощью мыши объявим в объекте вида (*class CPainterView*) переменные *StartPoint*, *EndPoint*, *OldPoint* класса *CPoint* для хранения координат начальной и конечной точки положения курсора при нажатии или отпускании клавиши мыши. Затем для выбора типа графического объекта и соответствующего цвета пера объявим переменные (флаги) *bLine*, *bDraw*, *bRectangle*, *bEllipse*, *bRed*, *bBlue*, *bGreen*, *bGray*, *bYellow*, *bWhite*, *bBlack* типа *Boolean*, а также целочисленную переменную *Pixel* для задания толщины пера:

```
class CPainterView : public CView {
        protected:
        CPainterView();
        DECLARE_DYNCREATE(CPainterView)
➧       CPoint StartPoint, EndPoint, OldPoint;
➧       boolean bLine, bDraw, bRectangle,bEllipse;
➧       boolean bRed,bBlue,bGreen,bGray,bYellow,bWhite,bBlack;
➧       int Pixel;
        // Attributes
        public:
        CPainterDoc* GetDocument() const;
➧       CPen newpen;
➧       CPen* oldpen;
➧       CBrush brush;
}
```

Для обработки сообщений от кнопок мыши при помощи *Class Wizard* добавим в объект вида методы *OnLButtonDown* и *OnLButtonUp*. В метод

*OnLButtonDown* добавим код сохранения начальной точки при нажатии левой клавиши мыши:

```
void CPainterView::OnLButtonDown(UINT nFlags, CPoint point) {
➪     StartPoint.x = point.x; StartPoint.y = point.y;
➪     OldPoint.x = StartPoint.x; OldPoint.y = StartPoint.y;
      CView::OnLButtonDown(nFlags, point);
}
```

При ее отпускании в метод *OnLButtonUp* добавим код сохранения конечной точки рисования:

```
void CPainterView::OnLButtonUp(UINT nFlags, CPoint point) {
➪     EndPoint.x = point.x;      EndPoint.y = point.y;
}
```

Мы получили положение начальной и конечной точки и можем нарисовать на экране графический объект типа эллипс, квадрат или линию. Например, нарисуем линию от начальных точек *StartPoint.x*, *StartPoint.y* до конечных точек *EndPoint.x*, *EndPoint.y*:

```
void CPainterView::OnLButtonUp(UINT nFlags, CPoint point) {
      EndPoint.x = point.x;      EndPoint.y = point.y;
➪     MoveTo(StartPoint.x,StartPoint.y);
➪     LineTo(EndPoint.x, EndPoint.y);
}
```

Для создания эластичных графических объектов типа резиновый контур при помощи *Class Wizard* добавим в объект вида метод *OnMouseMove*, в котором окно приложения будет «захватывать» мышь при получении сообщения о нажатии кнопки и «освобождать» – при ее отпускании:

```
void CPainterView::OnMouseMove(UINT nFlags, CPoint point) {
➪     if(nFlags&&MK_LBUTTON)
➪     { // код обработки сообщения }
}
```

### b) Создание интерфейса пользователя

С помощью мастера *AppWizard* создадим в проекте шаблон диалогового окна и включим в него все необходимые управляющие элементы. Для этого (как и в подпункте 3.5.3) выполним следующие шаги:

1. В редакторе ресурсов выберем *тип ресурса* → *Dialog* и *стиль* → *New*.



2. В шаблоне диалогового окна создадим два групповых поля. В первое поле для задания цвета пера включим семь переключателей типа *Radio Button*, а во второе поле для задания толщины пера включим четыре переключателя *Radio Button*.

3. Используя опцию *Properties*, для каждого управляющего элемента в поле *Caption* меняем название элементов: *Radio,..,Radio7* на *Red*, *Blue*, *Green*, *Gray*, *Yellow*,*White* и *Black* соответственно, а *Radio8,.., Radio11* – на *1 pixel*, *2 pixel*, *3 pixel* и *4 pixel*. На рис.32 показан разработанный шаблон диалогового окна.

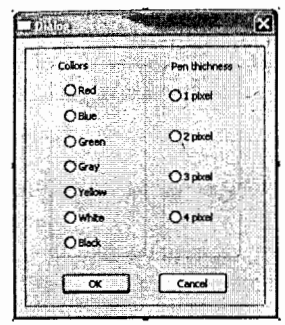


Рис.32. Диалоговое окно приложения *Painter*

4. Объявим класс диалогового окна и включим его в разрабатываемый проект. Мастер установки *AppWizard* создал базовый код этого класса (файлы *Dlg.cpp* и *Dlg.h*) и добавил его к проекту.

5. В этом классе переключатели *Radio1,..,Radio11* свяжем с обработчиком сообщений *BN_CLICKED*. Мастер установки *AppWizard* создал методы *OnBnClickedRadio*, в которые мы передадим соответствующие значения флагов (*Red*, *Blue*, *Green*, *Gray*, *Yellow*, *White*, *Black*, *Pixel Red*, *Blue*, *Green*, *Gray*, *Yellow*, *White*, *Black*, *Pixel*):

```
void CDlg::OnBnClickedRadio1() {
➪     Red=true; Blue=Green=Gray=Yellow=White=Black=false;
}
void CDlg::OnBnClickedRadio2() {
➪     Blue=true; Red=Green=Gray=Yellow=White=Black=false;
}
void CDlg::OnBnClickedRadio3() {
➪     Green=true; Red=Blue=Gray=Yellow=White=Black=false;
```

```
}
void CDlg::OnBnClickedRadio4() {
➡       Gray=true; Red=Green=Blue=Yellow=White=Black=false;
}
void CDlg::OnBnClickedRadio5() {
➡       Yellow=true; Red=Green=Gray=Blue=White=Black=false;
}
void CDlg::OnBnClickedRadio6() {
➡       bWhite=true; bRed=bGreen=bGray=bYellow=bBlue=bBlack=false;
}
void CDlg::OnBnClickedRadio7() {
➡       Black=true; Red=Green=Gray=Yellow=White=Blue=false;
}
void CDlg::OnBnClickedRadio8() {
➡       m_pixel=1;
}
void CDlg::OnBnClickedRadio9() {
➡       m_pixel=2;
}
void CDlg::OnBnClickedRadio10() {
➡       m_pixel=3;
}
void CDlg::OnBnClickedRadio11() {
➡       m_pixel=4;
}
```

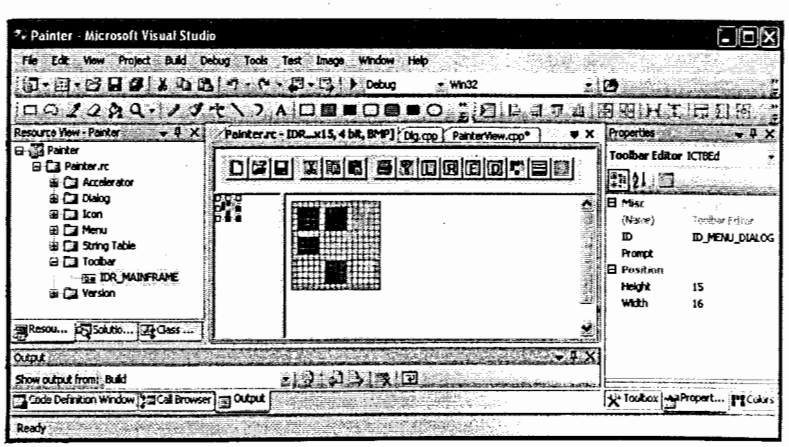


Рис.33. Создание управляющих элементов типа кнопка



Для выбора типа рисования графического объекта в *Tollbar* (рис.33) создадим шесть стандартных управляющих элементов типа кнопка: линия (*L*), прямоугольник (*R*), эллипс (*E*), произвольный объект (*D*), цвет и толщина пера. С помощью редактора ресурсов в строке *Menu* главного окна создадим команды управления (*Line, Rectangle, Ellipse, Draw, Dialog*) и свяжем их в классе *CPainterView* с обработчиком сообщений *COMMAND*. Затем в созданные мастером *AppWizard* методы передадим соответствующие значения флагов:

```
CPainterView::OnMenuLine() {
➡       bRectangle = false; bDraw = false; bEllipse = false; bLine = true;
}
void CPainterView::OnMenuEllipse() {
➡       bLine = false; bRectangle = false; bDraw = false; bEllipse = true;
}
void CPainterView::OnMenuDraw() {
➡       bLine = false; bRectangle = false; bEllipse = false; bDraw = true;
}
void CPainterView::OnMenuRectangle() {
➡       bLine = false; bEllipse = false; bDraw = false; bRectangle = true;
}
void CPainterView::OnMenuDialog() {
// код обработки диалогового окна
}
```

Кнопки выбора цвета и толщины пера свяжем с идентификатором диалогового окна *ID_MENU_DIALOG* (рис.33), а в методе *OnMenuDialog* передадим флагам выбора цвета и толщины пера соответствующие значения переменных из диалогового окна:

```
void CPainterView:: OnMenuDialog() {
        CDlg dlg;
        if ( dlg.DoModal() == IDOK ) {
        ➡       bRed    = dlg.Red;          bBlue   = dlg.Blue;
        ➡       bGreen = dlg.Green;         bGray   = dlg.Gray;
        ➡       bYellow = dlg.Yellow;       bWhite  = dlg.White;
        ➡       bBlack  = dlg.Black;        Pixel   = dlg.m_pixel;
        }
}
```

На этом разработка интерфейса пользователя, при помощи которого мы будем изменять параметры рисования графических объектов, закончена.

### c) Создание эластичных графических объектов

Для создания с помощью мыши эластичных графических объектов после получения сообщения о нажатии клавиши в методе *OnMouseMove* необходимо стереть предшествующее изображение графического объекта. Затем, получив новые координаты позиции мыши, надо перерисовать этот объект, используя полученные координаты. Для стирания текущего изображения объекта в контексте устройства устанавливается режим *SetROP2(R2_NOT)*. В этом режиме повторный вызов функции вывода объекта приводит к восстановлению состояния клиентской области окна, которое было перед первым его выводом.

Чтобы рисуемые объекты были прозрачными (т. е. незакрашенными), выберем в контекст устройства пустую кисть (*NULL_BRUSH*). Например, создание эластичного прямоугольника выполним следующим образом:

```
if((nFlags&&MK_LBUTTON)&&bRectangle) {
// Создадим указатель на старое перо и вызовем режим GetROP2
        nOldMode = pDC->GetROP2();
// Установим режим рисования без пера
        pDC ->SetROP2(R2_NOT);
// Выберем пустую кисть
        pDC->SelectStockObject(NULL_BRUSH);
// Сотрем предшествующее изображение графического объекта
        pDC->Rectangle(StartPoint.x,StartPoint.y,OldPoint.x,OldPoint.y);
// Перерисуем изображение объекта с новыми координатами
        pDC->Rectangle(StartPoint.x,StartPoint.y,point.x,point.y);
        OldPoint.x = point.x;
        OldPoint.y = point.y;
// Восстановим старый режим рисования
        pDC ->SetROP2(nOldMode);
}
```

Аналогично выполняется процедура для рисования линии и эллипса. На рис.34 показано окно приложения *Painter* с созданными различными графическими объектами, а ниже приведен листинг файла объекта вида для данного приложения.



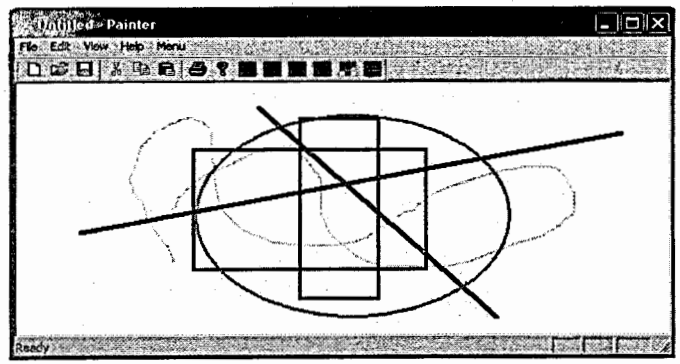


Рис.34. Окно приложения *Painter* с графическими объектами

```
// PainterView.cpp
#include "stdafx.h"
#include "Painter.h"
#include "PainterDoc.h"
#include "PainterView.h"
#include "Dlg.h"
#ifdef _DEBUG
#define new DEBUG_NEW
#endif
// CPainterView
IMPLEMENT_DYNCREATE(CPainterView, CView)
BEGIN_MESSAGE_MAP(CPainterView, CView)
        // Standard printing commands
        ON_COMMAND(ID_FILE_PRINT, &CView::OnFilePrint)
        ON_COMMAND(ID_FILE_PRINT_DIRECT, &CView::OnFilePrint)
        ON_COMMAND(ID_FILE_PRINT_PREVIEW, &CView::OnFilePrintPreview)
        ON_WM_LBUTTONDOWN()
        ON_WM_LBUTTONUP()
        ON_WM_MOUSEMOVE()
        ON_COMMAND(ID_MENU_LINE, &CPainterView::OnMenuLine)
        ON_COMMAND(ID_MENU_ELLIPSE, &CPainterView::OnMenuEllipse)
        ON_COMMAND(ID_MENU_DRAW, &CPainterView::OnMenuDraw)
        ON_COMMAND(ID_MENU_RECTANGLE, &CPainterView::OnMenuRectangle)
        ON_COMMAND(ID_MENU_DIALOG, &CPainterView::OnMenuDialog)
&CPainterView::OnMenuCollorsPen)
END_MESSAGE_MAP()
// CPainterView construction/destruction
```

```
CPainterView::CPainterView() {
➡       bLine = Rectangle = bEllipse = false;
➡       bDraw = true; Pixel =1;
}
CPainterView::~CPainterView() {
}
BOOL CPainterView::PreCreateWindow(CREATESTRUCT& cs) {
        return CView::PreCreateWindow(cs);
}
// CPainterView drawing
void CPainterView::OnDraw(CDC* /*pDC*/) {
        CPainterDoc* pDoc = GetDocument();
        ASSERT_VALID(pDoc);
        if (!pDoc)
        return;
}
// CPainterView printing
BOOL CPainterView::OnPreparePrinting(CPrintInfo* pInfo) {
        return DoPreparePrinting(pInfo);
}
void CPainterView::OnBeginPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/) {
}
void CPainterView::OnEndPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/) {
}
// CPainterView diagnostics
#ifdef _DEBUG
void CPainterView::AssertValid() const {
        CView::AssertValid();
}
void CPainterView::Dump(CDumpContext& dc) const {
        CView::Dump(dc);
}
CPainterDoc* CPainterView::GetDocument() const // non-debug version is inline {
        ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CPainterDoc)));
        return (CPainterDoc*)m_pDocument;
}
#endif //_DEBUG
// CPainterView message handlers
void CPainterView::OnLButtonDown(UINT nFlags, CPoint point) {
➡       StartPoint.x = point.x;
```



```
➡       StartPoint.y = point.y;
➡       OldPoint.x = StartPoint.x;
➡       OldPoint.y = StartPoint.y;
        CView::OnLButtonDown(nFlags, point);
}
void CPainterView::OnLButtonUp(UINT nFlags, CPoint point) {
➡       EndPoint.x = point.x;
➡       EndPoint.y = point.y;
➡       CClientDC*  pDC = new CClientDC(this);
        ➡       if(bRed) {newpen.CreatePen(PS_SOLID, Pixel, RGB(255, 0, 0));
                        oldpen = pDC->SelectObject(&newpen) ; }
        ➡       if(bBlue) {newpen.CreatePen(PS_SOLID, Pixel, RGB(0,0,255));
                        oldpen = pDC->SelectObject(&newpen) ; }
        ➡       if(bGreen) {newpen.CreatePen(PS_SOLID, Pixel, RGB(0,255, 0));
                        oldpen = pDC->SelectObject(&newpen) ; }
        ➡       if(bGray) {newpen.CreatePen(PS_SOLID, Pixel, RGB(127,127,127));
                        oldpen = pDC->SelectObject(&newpen) ; }
        ➡       if(bYellow) {newpen.CreatePen(PS_SOLID, Pixel, RGB(255,255,0));
                        oldpen = pDC->SelectObject(&newpen) ; }
        ➡       if(bWhite) {newpen.CreatePen(PS_SOLID, Pixel, RGB(255,255,255));
                        oldpen = pDC->SelectObject(&newpen) ; }
        ➡       if(bBlack) {newpen.CreatePen(PS_SOLID, Pixel, RGB(0,0,0));
                        oldpen = pDC->SelectObject(&newpen) ; }
➡       if(bLine) {
        pDC->MoveTo(StartPoint.x,StartPoint.y);
        pDC->LineTo(EndPoint.x,EndPoint.y); }
➡       if(bRectangle) {
        pDC->SelectStockObject(NULL_BRUSH);
        pDC->Rectangle(StartPoint.x,StartPoint.y,point.x,point.y); }
➡       if(bEllipse) {
        pDC->SelectStockObject(NULL_BRUSH);
        pDC->Ellipse(StartPoint.x,StartPoint.y,point.x,point.y);  }
        pDC->SelectObject(oldpen);
        newpen.DeleteObject() ;
        CView::OnLButtonUp(nFlags, point);
}
void CPainterView::OnMouseMove(UINT nFlags, CPoint point) {
➡       int nOldMode;
➡       CClientDC*  pDC = new CClientDC(this);
➡       if((nFlags&&MK_LBUTTON)&&bDraw) {
```

```
        if(bRed){newpen.CreatePen(PS_SOLID, Pixel, RGB(255, 0, 0));
                oldpen = pDC->SelectObject(&newpen); }
        if(bBlue) {newpen.CreatePen(PS_SOLID, Pixel, RGB(0,0,255));
                oldpen = pDC->SelectObject(&newpen) ; }
        if(bGreen) {newpen.CreatePen(PS_SOLID, Pixel, RGB(0,255, 0));
                oldpen = pDC->SelectObject(&newpen) ; }
        if(bGray) {newpen.CreatePen(PS_SOLID, Pixel, RGB(127,127,127));
                oldpen = pDC->SelectObject(&newpen) ; }
        if(bYellow) {newpen.CreatePen(PS_SOLID, Pixel, RGB(255,255,0));
                oldpen = pDC->SelectObject(&newpen) ; }
        if(bWhite){newpen.CreatePen(PS_SOLID, Pixel, RGB(255,255,255));
                oldpen = pDC->SelectObject(&newpen) ;}
        if(bBlack){newpen.CreatePen(PS_SOLID, Pixel, RGB(0,0,0));
                oldpen = pDC->SelectObject(&newpen) ; }
    pDC->MoveTo(StartPoint.x,StartPoint.y); pDC->LineTo(point.x,point.y);
    StartPoint.x = point.x; StartPoint.y = point.y;
    pDC->SelectObject(oldpen);
    newpen.DeleteObject() ;
    }
    if((nFlags&&MK_LBUTTON)&&bLine) {
        nOldMode = pDC->GetROP2();
        pDC ->SetROP2(R2_NOT);
        pDC->MoveTo(StartPoint.x,StartPoint.y);
        pDC->LineTo(OldPoint.x,OldPoint.y);
        pDC->MoveTo(StartPoint.x,StartPoint.y);
        pDC->LineTo(point.x,point.y);
        OldPoint.x = point.x; OldPoint.y = point.y;
        pDC ->SetROP2(nOldMode);
    }
    if((nFlags&&MK_LBUTTON)&&bRectangle) {
        nOldMode = pDC->GetROP2();
        pDC ->SetROP2(R2_NOT);
        pDC->SelectStockObject(NULL_BRUSH);
        pDC->Rectangle(StartPoint.x,StartPoint.y,OldPoint.x,OldPoint.y);
        pDC->Rectangle(StartPoint.x,StartPoint.y,point.x,point.y);
        OldPoint.x = point.x; OldPoint.y = point.y;
        pDC ->SetROP2(nOldMode);
    }
    if((nFlags&&MK_LBUTTON)&&bEllipse) {
        nOldMode = pDC->GetROP2();
```



```
        pDC ->SetROP2(R2_NOT);
        pDC->SelectStockObject(NULL_BRUSH);
        pDC->Ellipse(StartPoint.x,StartPoint.y,OldPoint.x,OldPoint.y);
        pDC->Ellipse(StartPoint.x,StartPoint.y,point.x,point.y);
        OldPoint.x = point.x; OldPoint.y = point.y;
        pDC ->SetROP2(nOldMode);
    }
    delete pDC;
    CView::OnMouseMove(nFlags, point);
}
void CPainterView::OnMenuLine() {
    bLine = true;
    bRectangle = bDraw = bEllipse = false;
}
void CPainterView::OnMenuEllipse() {
    bEllipse = true;
    bRectangle = bDraw = bLine = false;
}
void CPainterView::OnMenuDraw() {
    bDraw = true;
    bRectangle = bEllipse = bLine = false;
}
void CPainterView::OnMenuRectangle() {
    bRectangle = true;
bDraw = bEllipse = bLine = false;
}
void CPainterView:: OnMenuDialog() {
    CDlg dlg;
    if ( dlg.DoModal() == IDOK ) {
        bRed    = dlg.bRed;      bBlue   = dlg.bBlue;
        bGreen = dlg.bGreen;     bGray   = dlg.bGray;
        bYellow = dlg.bYellow;   bWhite  = dlg.bWhite;
        bBlack  = dlg.bBlack;    Pixel    = dlg.m_pixel;
    }
}
```

## 3.7. Разработка MFC-приложений для работы с файлами

### 3.7.1. Запись в метафайл и отображение графического изображения

Метафайлом называется хранящийся в памяти специальный объект, поддерживающий *собственный* контекст устройства. Поэтому все операции,

выполняемые с контекстом устройства, можно записать в метафайл и при необходимости воспроизвести (считать) из данного метафайла. В результате повторятся все графические операции, т. е. на экране автоматически создастся записанное в метафайле изображение.

### a) Создание объекта метафайла в MFC-приложении

Рассмотрим создание объекта метафайл в MFC-приложении. За основу нашего приложения возьмем уже созданный в подпункте 3.6.2 проект *Painter* и внесем в него необходимые добавления с тем, чтобы пользователь смог записывать графическую информацию в метафайл и считывать ее из метафайла. Сначала включим в заголовочный файл документа (*PainterDoc.h*) указатель на контекст устройства метафайла, который будет использоваться в программе:

```
// PainterDoc.h
class CPainterDoc : public CDocument {
public:
        virtual ~CPainterDoc();
➧       CMetaFileDC* pMetaFileDC;
}
```

Затем создадим объект *pMetaFileDC* и реализуем его в конструкторе объекта *документ*, вызывая метод *Create*:

```
// PainterDoc.cpp
// CPainterDoc construction/destruction
CPainterDoc::CPainterDoc() {
➧       pMetaFileDC = new CMetaFileDC();
➧       pMetaFileDC->Create();
}
```

Итак, мы создали контекст устройства метафайла, в котором будем дублировать все операции, выполняемые в клиентской области окна.

### b) Запись графического объекта в метафайл данных

Все, что пользователь рисует в клиентской области окна (на экране), должно дублироваться в метафайле. Это означает, что при каждом вызове метода для текущего контекста устройства необходимо вызвать аналогичный



метод для контекста метафайла. Для этого в методе *OnLButtonUp* вызываем контекст метафайла и копируем в него все, что рисуется в клиентской области окна проекта *Painter*:

```
void CPainterView::OnLButtonUp(UINT nFlags, CPoint point) {
        CPainterDoc* pDoc = GetDocument();
        ASSERT_VALID(pDoc);
        EndPoint.x = point.x;
        EndPoint.y = point.y;
        CClientDC*  pDC = new CClientDC(this);
        if(bRed){newpen.CreatePen(PS_SOLID, Pixel, RGB(255, 0, 0));
                        oldpen = pDC->SelectObject(&newpen) ;
                ➧       pDoc->pMetaFileDC->SelectObject(&newpen) ;
        }
        if(bBlue){newpen.CreatePen(PS_SOLID, Pixel, RGB(0,0,255));
                        oldpen = pDC->SelectObject(&newpen) ;
                ➧       pDoc->pMetaFileDC->SelectObject(&newpen) ;
        }
        if(bGreen){newpen.CreatePen(PS_SOLID, Pixel, RGB(0,255, 0));
                        oldpen = pDC->SelectObject(&newpen) ;
                ➧       pDoc->pMetaFileDC->SelectObject(&newpen) ;
        }
        if(bGray){newpen.CreatePen(PS_SOLID, Pixel, RGB(127,127,127));
                        oldpen = pDC->SelectObject(&newpen) ;
                ➧       pDoc->pMetaFileDC->SelectObject(&newpen) ;
        }
        if(bYellow){newpen.CreatePen(PS_SOLID, Pixel, RGB(255,255,0));
                        oldpen = pDC->SelectObject(&newpen) ;
                ➧       pDoc->pMetaFileDC->SelectObject(&newpen) ;
        }
        if(bWhite){newpen.CreatePen(PS_SOLID, Pixel, RGB(255,255,255));
                        oldpen = pDC->SelectObject(&newpen) ;
                ➧       pDoc->pMetaFileDC->SelectObject(&newpen) ;
        }
        if(bBlack){newpen.CreatePen(PS_SOLID, Pixel, RGB(0,0,0));
                        oldpen = pDC->SelectObject(&newpen) ;
                ➧       pDoc->pMetaFileDC->SelectObject(&newpen) ;
        }
        if(bLine) { pDC->MoveTo(StartPoint.x,StartPoint.y);
➧       pDoc->pMetaFileDC->MoveTo(StartPoint.x,StartPoint.y);
```

```
        pDC->LineTo(EndPoint.x,EndPoint.y);
➡       pDoc->pMetaFileDC->LineTo(EndPoint.x,EndPoint.y);
        }
        if(bRectangle) {
        pDC->SelectStockObject(NULL_BRUSH);
➡       pDoc->pMetaFileDC->SelectStockObject(NULL_BRUSH);
        pDC->Rectangle(StartPoint.x,StartPoint.y,point.x,point.y);
➡       pDoc->pMetaFileDC->Rectangle(StartPoint.x,StartPoint.y,point.x,point.y);
        }
        if(bEllipse) { pDC->SelectStockObject(NULL_BRUSH);
➡       pDoc->pMetaFileDC->SelectStockObject(NULL_BRUSH);
        pDC->Ellipse(StartPoint.x,StartPoint.y,point.x,point.y);
➡       pDoc->pMetaFileDC->Ellipse(StartPoint.x,StartPoint.y,point.x,point.y);
        }
        pDC->SelectObject(oldpen);
➡       pDoc->pMetaFileDC->SelectObject(oldpen);
        newpen.DeleteObject() ;
        CView::OnLButtonUp(nFlags, point);
}
```

### c) Отображение графического объекта из метафайла данных

Для того чтобы в методе *OnDraw* воспроизвести содержимое метафайла, сначала надо закрыть старый метафайл и получить его логический номер, а затем воспроизвести новый метафайл по данному логическому номеру:

```
void CPainterView::OnDraw(CDC* /*pDC*/) {
        CPainterDoc* pDoc = GetDocument();
        ASSERT_VALID(pDoc);
        if (!pDoc)
        return;
        CDC* pDC = GetDC();
// Закрываем старый метафайл
➡       HMETAFILE MetaFileHandle = pDoc ->pMetaFileDC->Close();
➡       pDC->PlayMetaFile(MetaFileHandle);
```

Однако после закрытия метафайла мы уже не сможем записать в него информацию. Для этого нужно создать новый метафайл и воспроизвести в нем старый (для которого у нас есть логический номер):

```
// Создадим новый метафайл и воспроизведем в нём старый
➡       CMetaFileDC* ReplasementMetaFile = new CMetaFileDC();
```



```
➡       ReplasementMetaFile->Create();
➡       ReplasementMetaFile->PlayMetaFile(MetaFileHandle);
// Заменим старый метафайл новым и удалим старый
➡       DeleteMetaFile(MetaFileHandle);
➡       delete pDoc ->pMetaFileDC;
➡       pDoc ->pMetaFileDC = ReplasementMetaFile;
}
```

Таким образом, в методе *OnDraw* мы воспроизвели графическое отображение из старого метафайла в клиентской области окна и создали новый метафайл для следующего обновления.

### d) Запись данных метафайла на диск и вывод графического изображения из файла в клиентскую область окна

Для сохранения данных метафайла на диск воспользуемся методом *CopyMetaFile* класса *CMetaFileDC*. При помощи мастера *ClassWizard* свяжем идентификаторы *ID_FILE_SAVE* и *ID_FILE_OPEN* (команд меню *Save* и *Open*) с методом обработки *COMMAND*. С помощью этих команд меню мы будем сохранять изображение в метафайле *Painter.wmf* и загружать его в программу из сохраненного ранее метафайла. Для этого закроем метафайл в методе *OnFileSave*, получим его логический номер и передадим его вместе с именем файла методу *CopyMetaFile*, в котором будем хранить содержимое метафайла:

```
void CPainterView::OnFileSave() {
        CPainterDoc* pDoc = GetDocument();
➡       HMETAFILE MetaFileHandle = pDoc->pMetaFileDC->Close();
➡       CopyMetaFile(MetaFileHandle, " painter.wmf ");
```

Далее, создадим новый метафайл и воспроизведем в нем старый:

```
void CPainterView::OnFileSave() {
        CPainterDoc* pDoc = GetDocument();
        ASSERT_VALID(pDoc);
        HMETAFILE MetaFileHandle = pDoc->pMetaFileDC->Close();
        CopyMetaFile(MetaFileHandle, " painter.wmf ").
➡       CMetaFileDC* ReplasementMetaFile = new CMetaFileDC();
➡       ReplasementMetaFile->Create();
➡       ReplasementMetaFile ->PlayMetaFile(MetaFileHandle);
```

Затем заменим старый метафайл новым и удалим старый метафайл:

```
void CPainterView::OnFileSave() {
        CPainterDoc* pDoc = GetDocument();
        ASSERT_VALID(pDoc);
        HMETAFILE MetaFileHandle = pDoc->pMetaFileDC->Close();
        CopyMetaFile(MetaFileHandle, "graph.wmf").
        CMetaFileDC* ReplasementMetaFile = new CMetaFileDC();
        ReplasementMetaFile->Create();
        ReplasementMetaFile->PlayMetaFile(MetaFileHandle).
➧       DeleteMetaFile(MetaFileHandle);
➧       delete pDoc ->pMetaFileDC;
➧       pDoc ->pMetaFileDC = ReplasementMetaFile;
}
```

Итак, мы сохранили графическое изображение объектов в клиентской области окна программы на диске (файл *Painter.wmf*). Теперь создадим код программы для загрузки графического образа из метафайла с диска и вывода его в клиентскую область окна. Загрузку файла *Painter.wmf* выполним с помощью метода *GetMetaFile*. Для этого создадим новый объект класса *pMetaFileDC* и воспроизведем в нем метафайл, загружаемый с диска, а потом заменим старый метафайл новым:

```
void CPainterView::OnFileOpen() {
        CPainterDoc* pDoc = GetDocument();
➧       HMETAFILE MetaFileHandle = GetMetaFile("painter.wmf");
// Создадим объект класса pMetaFileDC и воспроизведем в нем метафайл
➧       CMetaFileDC* ReplasementMetaFile = new CMetaFileDC();
➧       ReplasementMetaFile->Create();
➧       ReplasementMetaFile->PlayMetaFile(MetaFileHandle);
// Заменим старый метафайл новым
➧       DeleteMetaFile(MetaFileHandle);
➧       delete pDoc ->pMetaFileDC;
➧       pDoc ->pMetaFileDC = ReplasementMetaFile;
// Отобразим новый метафайл в объекте вида в методе OnDraw
    ➧   Invalidate();
}
```

Таким образом, разработанная программа обеспечивает сохранение графического изображения на диске и позволяет загружать его с диска.



### 3.7.2. Сериализация объектов в MFC-приложении

#### а) Сериализация на диск стандартных объектов MFC-приложения

Сериализацией называется процесс записи или чтения объекта с диска. В MFC-приложениях вся работа с данными происходит в объекте *документ*, где мастер установки *AppWizard* при создании автоматически включил метод *Serialize* и начальный базовый код для сериализации данных:

```
void CTextDoc::Serialize(CArchive& ar) {
        if (ar.IsStoring())
                {
        ➧       // Добавить код записи объекта в файл
                }
        else
                {
        ➧       //Добавить код считывания объекта из файла
                }
}
```

Нам в программе только осталось добавить исполняемый код программы.

Рассмотрим процесс создания исполняемого кода на примере уже созданного в подпункте 3.4.3 приложения *Text*, в котором данные вводились в виде символов с клавиатуры и затем отображались в клиентской области окна. Для записи на диск в файл стандартного объекта *StringData* класса *CString*, где хранятся символы, полученные с клавиатуры, мы вызываем метод *IsStoring* и в объект *ar* класса *CArchiv* (по аналогии работы с потоками *cout* и *cin* в C++) записываем объект *StringData* или считываем в него содержимое из *ar*:

```
void CTextDoc::Serialize(CArchive& ar) {
        if (ar.IsStoring()) {
        ➧       ar << StringData;
        }
        Else {
        ➧       ar >> StringData;
        }
}
```

Теперь нам осталось только сообщить приложению об изменении данных. Выполним это с помощью вызова в методе *OnChar* функции *SetModifiedFlag*:

```
void CTextView::OnChar(UINT nChar, UINT nRepCnt, UINT nFlags) {
        CTextDoc* pDoc = GetDocument();
        ASSERT_VALID(pDoc);
        if (!pDoc)
        return;
        pDoc->StringData += (char)nChar;
        Invalidate();
➧       pDoc->SetModifiedFlag();
        CView::OnChar(nChar, nRepCnt, nFlags);
}
```

Программа готова, она записывает стандартный объект *StringData* в файл и считывает содержимое из файла в этот объект. Когда выбирается команда *File → Save As*, то данные из объекта записываются в файл, например, *Data.dat*. После сохранения данных можно считать этот файл командой *File → Open*. При загрузке файла и изменении документа объект *вид* объявляется недействительным, поэтому происходит его автоматическое обновление с выводом нового документа, т. е. с новым содержимым *StringData*.

### b) Сериализация на диск нестандартных объектов приложения

Рассмотрим процедуру сериализации нестандартных объектов в MFC-приложении. Для этого в приложении *Text* объявим новый класс *CData*, который содержит объект *Data* класса *CString*, и три метода для работы с этим объектом: *AddText* – для добавления текста в конец строки, *DrawText* – для вывода текста в текущий контекст устройства и *ClearText* – для очистки содержимого строки. Здесь же определим код этих методов:

```
class CData : public CObject {
    private:
        ➧ CString Data;
    public:
        ➧ CData() {Data = CString(" "); }
        ➧ void AddText(CString text) {Data += text;}
        ➧ void DrawText(CDC* pDC) {pDC->TextOut(0,0,Data);}
        ➧ void ClearText() {Data = " "; }
}
```



Затем включим в заголовочный файл объекта *документ* ссылку на новый класс, чтобы *документ* знал о существовании класса *CData*, и там же создадим объект *DataObject* класса *CData*:

```
// TextDoc.h : interface of the CTextDoc class
#include "Data.h"
class CTextDoc : public CDocument {
protected:
        CTextDoc();
        DECLARE_DYNCREATE(CTextDoc)
public:
➧       CData DataObject;
}
```

Вводимые с клавиатуры символы (в отличие от кода программы по сериализации стандартных объектов) будем прямо добавлять в методе *OnChar* в объект *DataObject*:

```
void CTextView::OnChar(UINT nChar, UINT nRepCnt, UINT nFlags) {
        CTextDoc* pDoc = GetDocument();
➧       pDoc->DataObject.AddText(CString (nChar));
        Invalidate();
        CView::OnChar(nChar, nRepCnt, nFlags);
}
```

А содержимое объекта *Data* выведем на экран при помощи функции *DrawText* в методе *OnDraw*:

```
void CTextView::OnDraw(CDC* /*pDC*/) {
        CTextDoc* pDoc = GetDocument();
        ASSERT_VALID(pDoc);
        if (!pDoc)
        return;
        CDC* pDC = GetDC();
➧       pDoc->DataObject.DrawText(pDC);
}
```

Так как стандартные операторы C++ типа *cout* и *cin* для работы с нестандартными объектами недопустимы, то создадим код программы нашего метода сериализации. Для этого нам необходимо:

1. Включить в определение класса *CData* макрос *DECLARE_SERIAL*, который объявляет методы, используемые в процессе сериализации:

```
class CData : public CObject {
        private:
            CString data;
            DECLARE_SERIAL(CData);// макрос
        public:
            CData() {data = CString(" "); }
            void AddText(CString text) {data += text;}
            void DrawText(CDC* pDC) {pDC->TextOut(0,0,data);}
            void ClearText() {data = " "; }
}
```

2. Переопределить в файле *CData.cpp* метод *Serialize(CArchive& archive)* класса *CObject*, и запрограммировать сериализацию объекта:

```
#include "stdafx.h"
#include "TextDoc.h"
void CData::Serialize(CArchive& archive) {
        CObject::Serialize(archive);        // вызовем базовый класс
        if(archive.IsStoring())
            {
            archive << data;                // сериализация записи объекта
        }
        else
        {
            archive >> data;                // сериализация чтения объекта
        }
}
IMPLEMENT_SERIAL(CData,CObject,0);
```

3. Добавить макрос *MPLEMENT_SERIAL*, который содержит дополнительные методы, используемые в Visual C++ для сериализации.

4. Выполнить сериализацию нашего объекта *DataObject*, вызвав для этого метод *Serialize* внутри метода *Serialize* в объекте *документ*:

```
 // CTextDoc serialization
 void CTextDoc::Serialize(CArchive& ar) {
        DataObject.Serialize(ar);
 }
```



Итак, мы создали код программы, которая осуществляет сериализацию нестандартных объектов в MFC-приложении.

### 3.7.3. Работа с файлами в MFC-приложении, запись и считывание данных с диска

Разработаем новое приложение *Graph* для вывода графика функций $F_1 = \sin(k \times x) \times A_1$, $F_2 = \cos(m \times x) \times A_2$ в клиентскую область окна и с вводом параметров функций в диалоговом окне. На базе этого приложения рассмотрим процедуру создания кода программы для чтения и записи данных на диск в MFC-приложении.

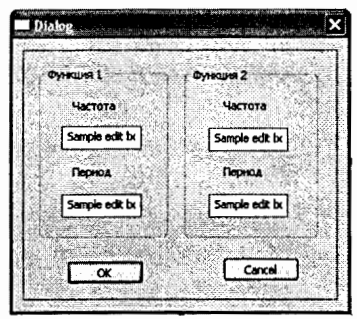


Рис.35. Шаблон диалогового окна в приложении *Graph*

Процедура создания приложения *Graph* для чтения и записи данных на диск состоит из следующих шагов:

1. При помощи мастера установки *AppWizard* создадим новый проект *Graph.*

2. В редакторе ресурсов создадим шаблон диалогового окна (рис.35) для ввода в программу параметров функций (периода и амплитуды).

3. Объявим класс диалогового окна и включим его в разрабатываемый проект. Мастер установки *AppWizard* создал базовый код этого класса (файлы *Dlg.cpp* и *Dlg.h*) и добавил его к проекту.

4. С помощью редактора ресурсов создадим в строке главного окна команду управления *Menu* и свяжем ее с обработчиком сообщений *COMMAND*.

5. Создадим объект подкласса *CDialog* и вызовем у него функцию-член *DoModal* для создания модального диалогового окна.

6. Передадим в созданном методе *DoModal* соответствующие значения параметров функций из диалогового окна, используя для этого указатель *pDoc* на объект документа *pDoc->data1 = atoi(dlg.m_text1)*:

```
void CGraphView::OnMenu() {
➡       CGraphDoc* pDoc = GetDocument();
        ASSERT_VALID(pDoc);
        CDlg dlg;
        if ( dlg.DoModal() == IDOK ) {
        ➡       pDoc->data1 = atoi(dlg.m_text1);
        ➡       pDoc->data2 = atoi(dlg.m_text2);
        ➡       pDoc->data3 = atoi(dlg.m_text3);
        ➡       pDoc->data4 = atoi(dlg.m_text4);
        }
        Invalidate();
}
```

7. Создадим в приложении код программы для рисования графика функций и осей. Для этого в классе *CGraphView* (в заголовочном файле *GraphView.h*) объявляем два новых члена-функции: *void Functions*(), *void axis*(); типа *public*.

8. В структуре функции *Functions*, в методе *CClientDC*, выполним построение графиков функций $F_1 = \sin(x) \times A_1, F_2 = \cos(x) \times A_2$, а в структуре функции *axis* – построение осей координат:

```
➡ void CGraphView::Functions() {
// Создадим графики функции  sin и cos
        CGraphDoc* pDoc = GetDocument();
        CClientDC* pDC = new CClientDC(this);
        pi = 3.14159265359; buf1[1]=60; buf1[2]=180;buf2[1]=60; buf2[2]=180;
        i=i1=j1=j2=0;
        for(x = 60; x < 780; x++) {
        i=i++;  j2=j2+pDoc->data3;  j1=j1+pDoc->data1;
        newpen.CreatePen(PS_SOLID, 3, RGB(0, 245, 0));
        oldpen = pDC->SelectObject(&newpen) ;
// Запишем данные графика sin в буфер buf3[i] и считаем данных из файла в буфер buf3[i]
        if(pDoc->data==1) {
        y1=(sin(j1*pi/180))*(pDoc->data2); buf3[i]=(int)y1; buf5[i]=x;
        }
        else
        {
```



```
        y1=buf3[i];
        }
        pDC->MoveTo(buf1[1], buf1[2]); pDC->LineTo(x,(int)y1+180); buf1[1]=x;
        buf1[2]=(int)y1+180;
        pDC->SelectObject(oldpen);
        newpen.DeleteObject() ;
        newpen.CreatePen(PS_SOLID, 3, RGB(245,0, 0));
        oldpen = pDC->SelectObject(&newpen) ;
// Запишем данные графика cos в буфер buf4[i] и считаем данные из файла в буфер buf4[i]
        if(pDoc->data==1) {
        y2=(cos(j2*pi/180))*pDoc->data4;  buf4[i]=(int)y2;
        }
        else
        {
        y2=buf4[i];
        }
        pDC->MoveTo(buf2[1], buf2[2]); pDC->LineTo(x,(int)y2+180);
        buf2[1]=x; buf2[2]=(int)y2+180;
        pDC->SelectObject(oldpen);
        newpen.DeleteObject() ;
        }
 delete pDC;
}

➡ void CGraphView::axis() {
// Создадим оси координат
        CClientDC* pDC = new CClientDC(this);
        newpen.CreatePen(PS_SOLID, 2, RGB(0,0, 0));
        oldpen = pDC->SelectObject(&newpen) ;
        pDC->MoveTo(60, 180); pDC->LineTo(800,180); pDC->MoveTo(60, 40);
        pDC->LineTo(60,250); pDC->SelectObject(oldpen);
        newpen.DeleteObject() ;
        pDC->TextOut (40,175, "0", 1); pDC->TextOut (35,125, "50", 2);
        pDC->TextOut (30,75, "100", 3); pDC->TextOut (30,225, "-50", 3);
        newpen.CreatePen(PS_DASH, 1,RGB(127,127, 127));
        oldpen = pDC->SelectObject(&newpen) ;
        pDC->MoveTo(60, 230); pDC->LineTo(780,230); pDC->MoveTo(60, 130);
        pDC->LineTo(780,130); pDC->MoveTo(60, 80); pDC->LineTo(780,80);
        pDC->MoveTo(240, 230); pDC->LineTo(240,80); pDC->MoveTo(420, 230);
        pDC->LineTo(420,80); pDC->MoveTo(600, 230); pDC->LineTo(600,80);
```

```
        pDC->MoveTo(780, 230); pDC->LineTo(780,80); pDC->SelectObject(oldpen);
        newpen.DeleteObject() ;
        delete pDC;
}
```

9. В методе *OnDraw* вызываем эти функции:

```
void CGraphView::OnDraw(CDC* /*pDC*/) {
        CGraphDoc* pDoc = GetDocument();
        ASSERT_VALID(pDoc);
        if (!pDoc)
        return;
        CDC* pDC = GetDC();
        pDC->TextOut(200,300,pDoc->data1_string);
        pDC->TextOut(200,320,pDoc->data2_string);
        pDC->TextOut(200,340,pDoc->data3_string);
        pDC->TextOut(200,360,pDoc->data4_string)
        pDC->TextOut(200,380,pDoc->data_string);
➧       axis();
➧       Functions();
}
```

На рис.36 показано окно приложения *Graph* для вывода графика функций в клиентскую область окна (на экран).

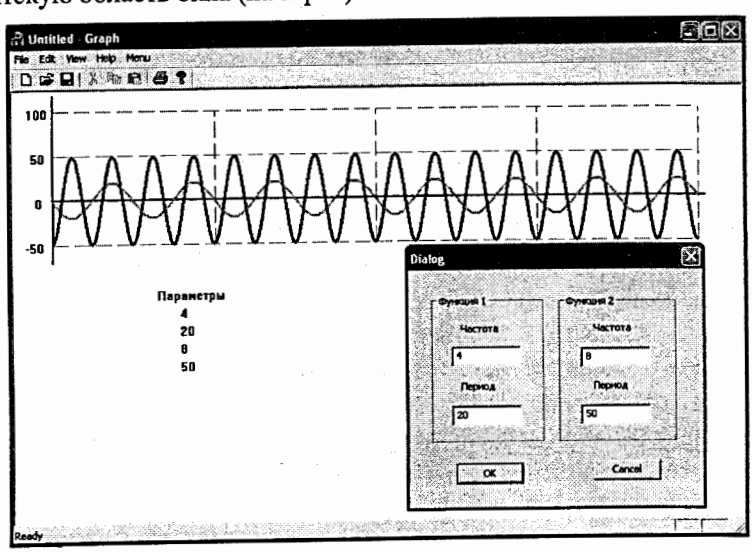


Рис.36. Окно приложения *Graph* для вывода графика функций



В приложении *Graph* для хранения значений функций $F_1 = \sin(x)$ и $F_2 = \cos(x)$ применяются массивы данных *buf3* и *buf4*, которые мы используем для чтения и записи информации на диск (в файл). Процедура создания кода программы для записи данных на диск состоит из следующих шагов:

1. При помощи *ClassWizard* свяжем идентификаторы *ID_FILE_SAVE*, *ID_FILE_OPEN* команд меню *Save* и *Open* с методом обработки *COMMAND*.

2. В обработчике сообщения *OnFileSave* создадим объект *OutFile* класса *CFile* (список методов класса *CFile* приведен в табл.11) и откроем его для записи данных:

```
// GraphView.cpp
void CGraphView::OnFileSave() {
➧       CFile File("data.dat",CFile::modeCreate | CFile::modeWrite);
➧       File.Close();
```

В методе *OnFileSave* мы передаем конструктору константы класса *CFile* → *modeCreate*, (т. е. мы сообщаем об открытии нового файла) и константу *modeWrite*, сообщающую, что файл создается для записи данных. В табл.12 приведен список методов и констант класса *CFile*, определяющих режим открытия файла.

Таблица 11. Перечень методов класса *CFile*

| | |
|---|---|
| *CFile* | Конструирует объект класса *CFile* по заданному пути или по файловому логическому номеру |
| *Close* | Закрывает файл и удаляет объект |
| *GetFileName* | Получает имя заданного файла |
| *GetFilePath* | Получает полный файловый путь для заданного файла |
| *Read* | Читает данные из файла с заданной позиции |
| *Remove* | Удаляет заданный файл |
| *Seek* | Перемещает указатель в заданную позицию |
| *SeekToBegin* | Перемещает указатель в начало файла |
| *SeekToEnd* | Перемещает указатель в конец файла |
| *SetFilePath* | Задает полный путь файла |
| *Write* | Записывает данные в файл с текущей позиции |

Таблица 12. Режимы открытия файла в классе *CFile*

| | |
|---|---|
| *CFile:: modeCreate* | Создает новый файл |
| *CFile:: modRead* | Создает файл только для чтения |
| *CFile:: modReadWrite* | Создает файл для чтения и записи |
| *CFile:: modWrite* | Создает файл только для записи |
| *CFile:: typeBinary* | Устанавливает двоичный режим |
| *CFile:: typeText* | Устанавливает текстовый режим со специальной обработкой пар символов конца перевода строки |

3. Файл *data.dat* открыт, и мы при помощи оператора цикла запишем в него значения функций в диапазоне углов от 0 до 720, которые хранятся в массивах *buf3*[i] и *buf4*[i]. Так как тип данных в массивах → *double*, то при помощи оператора *Format* производим преобразование типа данных и переводим метку указателя в начало файла → *SeekToBegin*:

```
void CGraphView::OnFileSave() {
CString s;
CFile File ("data.dat",CFile::modeCreate | CFile::modeWrite);
      for(int i=0;i<720;i++) {
          s.Format("%7d",buf4[i]);  File.Write(s,7);
          s.Format("%7d",buf3[i]);  File.Write(s,7);
          File.Seek(14* i, CFile::begin);
      }
      File.Close();
}
```

4. Для создания кода программы чтения строк из файла используем метод *ReadString* класса *CStdioFile*. Процедура чтения начинается с создания стандартной панели для выбора файла в меню → *Open* и отображения ее в модальном диалоговом окне с помощью метода *DoModal*:

```
void CGraphView::OnFileOpen() {
    // создаем стандартную панель выбора файла Open
CFileDialog  DlgOpen(TRUE,(LPCSTR)"dat", NULL,OFN_HIDEREADONLY, (LPCSTR)
"Text Files(*.dat) |*.dat||");
    // отображаем стандартную панель выбора файла Open в методе DoModal
      If (DlgOpen.DoModal()==IDOK) {
    // вставляем код программы чтения данных из файла
      }
};
```



5. Создадим в методе *DoModal* новый объект *File* класса *CFile*, который откроем в режиме чтения. Далее, при помощи оператора цикла считаем символьные строки из файла, произведем их преобразование в целочисленный тип и запишем их в массив *buf3* и *buf4*:

```
void CGraphView::OnFileOpen() {
      CGraphDoc* pDoc = GetDocument();
      ASSERT_VALID(pDoc);    pDoc->data=0;
      CString s;
      int i,m;
// Создадим стандартную панель выбора файла Open
      CFileDialog DlgOpen(TRUE,(LPCSTR)"dat",NULL,OFN_HIDEREADONLY,
      (LPCSTR)" Text Files (*.dat) |*.dat||");
    // Отобразим стандартную панель выбора файла Open в методе DoModal
      if(DlgOpen.DoModal()==IDOK) {
// Создадим объект и откроем файл для чтения
      CStdioFile File(DlgOpen.GetPathName(),CFile::modeRead|CFile::typeBinary);
      int j=0;
      for(i=0;i<720;i++) {
      // читаем строки из файла
            CString& ref=s;
            File.ReadString(ref); // передаем ссылку на строку s
            m=atoi(s); buf4[i]=m;
            j=j+1;
            File.Seek(7 * j, CFile::begin);
            File.ReadString(ref); // передаем ссылку на строку s
            m=atoi(s); buf3[i]=m;
            j=j+1;
            File.Seek(7 * j, CFile::begin);
            }
      File.Close();
      }
      Invalidate();
}
```

6. При помощи метода *Invalidate* обновим содержимое клиентской области окна, т. е. перерисуем графики функций, используя для этого данные из считываемого файла.

Разработанная программа *Graph* обеспечивает сохранение и запись данных на диск.

## Заключение

Конечно, одного учебного пособия такого объема явно недостаточно, чтобы полно изучить все средства *Microsoft Visual* C++ и библиотеки MFC. В данном курсе мы не рассмотрели создание Windows-приложений, имеющих *многооконный интерфейс*, приложений, которые могут одновременно работать с документами различных типов, а также возможности библиотеки MFC по работе с *базами данных*. Поэтому предлагаем продолжить изучение MFC, используя книги из списка рекомендованной литературы и книги по *Microsoft Visual* C++ из серии «Библиотека системного программиста».

## Рекомендуемая литература:


1. Давыдов В. Visual C++. Разработка Windows-приложений с помощью MFC и API-функций. СПб.: БХВ – Петербург, 2008.
2. С. Холзнер. Visual C++6. СПб.:Петербург, 2007.
3. Круглински Д., Уингоу С., Шеферд Дж. Программирование на *Microsoft* Visual C++ 6.0 для профессионалов. СПб.: Питер, 2000.
4. Павловская Т.А. C/C++ Программирование на языке высокого уровня. Учебник для вузов. СПб.: Питер, 2002.

Учебное издание

Калинников Владимир Александрович

**Разработка Windows-приложений с помощью MFC-библиотеки классов в среде программирования Microsoft Visual C++2008**

УНЦ-2010-45

Редактор *М. И. Зарубина*

Получено 31.08.2010. Подписано в печать 06.10.2010.
Формат 60 × 90/16. Усл. печ. л. 8,68. Уч.-изд. л. 8,0. Тираж 150. Заказ № 571

Издательский отдел Объединенного института ядерных исследований
141980, г. Дубна, Московская обл., ул. Жолио-Кюри, 6
E-mail: publish@jinr.ru
www.jinr.ru/publish/