Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Н. А. Кириенко

РАЗРАБОТКА WINDOWS-ПРИЛОЖЕНИЙ НА ЯЗЫКЕ C++ С ИСПОЛЬЗОВАНИЕМ БИБЛИОТЕКИ MFC

*Рекомендовано УМО по образованию
в области информатики и радиоэлектроники
в качестве учебно-методического пособия для специальности 1-40 01 02-02
«Информационные системы и технологии (в экономике)»
по дисциплине «Визуальные средства разработки
программных приложений»*

Минск БГУИР 2012

УДК 004.438(075.8)
ББК 32.973.26-018.1я73
К43

Р е ц е н з е н т ы:

кафедра прикладной информатики учреждения образования
«Белорусский государственный аграрный технический университет»
(протокол №7 от 14.02.2012 г.);

главный научный сотрудник
Объединенного института проблем информатики
Национальной академии наук Беларуси,
доктор технических наук А. А. Дудкин

Кириенко, Н. А.

К43 Разработка Windows-приложений на языке С++ с использованием библиотеки MFC : учеб.-метод. пособие / Н. А. Кириенко. – Минск : БГУИР, 2012. – 202 с. : ил.

ISBN 978-985-488-877-4.

Представлены теоретические материалы и практические примеры по основам разработки Windows-приложений на языке С++ с использованием библиотеки MFC. Примеры разработаны в среде визуального программирования Microsoft Developer Studio 2008. Предназначено для студентов специальности 1-40 01 02-02 «Информационные системы и технологии (в экономике)» всех форм обучения.

Для студентов специальности 1-40 01 02-02 «Информационные системы и технологии (в экономике)» по дисциплине «Визуальные средства разработки программных приложений».

УДК 004.438(075.8)
ББК 32.973.26-018.1я73

ISBN 978-985-488-877-4

Содержание

Введение

Дисциплина «Визуальные средства разработки программных приложений» позволяет студентам получить теоретические знания и практические навыки в области разработки прикладных приложений на основе средств визуального проектирования и программирования. Курс базируется на средствах визуального программирования на языке C++ в среде Microsoft Developer Studio (2008 и выше) с использованием библиотеки классов Microsoft Foundation Classes (MFC).

Visual C++ представляет собой мощный и сложный инструмент для создания Windows-приложений. Несмотря на то что объем и сложность разрабатываемых программ увеличиваются, для их создания от программиста требуется не больше, а меньше усилий, если он научился правильно выбирать инструментальные средства.

Среда разработки оснащена набором мастеров (Wizard), формирующих программный код. Этот продукт позволяет в считанные секунды создать вполне работоспособное приложение Windows. Включенная в состав Visual C++ библиотека классов MFC содержит практически весь программный интерфейс Windows и позволяет пользоваться при программировании средствами более высокого уровня, чем обычные вызовы функций.

Визуальные средства разработки интерфейса пользователя превращают процесс создания разнообразных меню, диалоговых окон, элементов управления в довольно увлекательный процесс выбора и настройки графических средств. Для этого используются мастера (или генераторы) приложений. Программист отвечает на вопросы мастера приложений и определяет свойства приложения: поддерживает ли оно многооконный режим, технологию OLE, трехмерные органы управления, справочную систему. Генератор приложений создаст приложение, отвечающее требованиям, и предоставит исходные тексты. Пользуясь им как шаблоном (или каркасом), программист сможет быстро разрабатывать свои приложения.

В настоящем учебно-методическом пособии представлены материалы, позволяющие освоить основные темы курса «Визуальные средства разработки программных приложений» и выполнить задания по программированию, предусмотренные учебной программой. Пособие является дополнением к существующим пособиям [1, 2] по курсу «Визуальные средства разработки программных приложений», в которых представлены методические материалы по выполнению лабораторных работ.

1. Инструментальные средства разработки приложений

Одним из основных визуальных средств разработки Windows-приложений на языке C++ является среда разработки Visual Studio 2005/2008/2010 [3, 4], интерфейс которых достаточно похож. Рассмотрим основные возможности данной среды программирования.

Среда разработки Visual Studio 2008 (VS) позволяет разрабатывать программы на нескольких языках программирования: C++, C#, Visual Basic и ASP.NET. При первом старте отображается окно, в котором среда попросит указать предпочитаемый язык программирования. От этого выбора зависит, какие типы файлов и проектов будут предлагаться при создании нового приложения.

После выбора языка программирования загружается главное окно среды разработки, изображенное на рис. 1.1, в котором представлены следующие разделы.

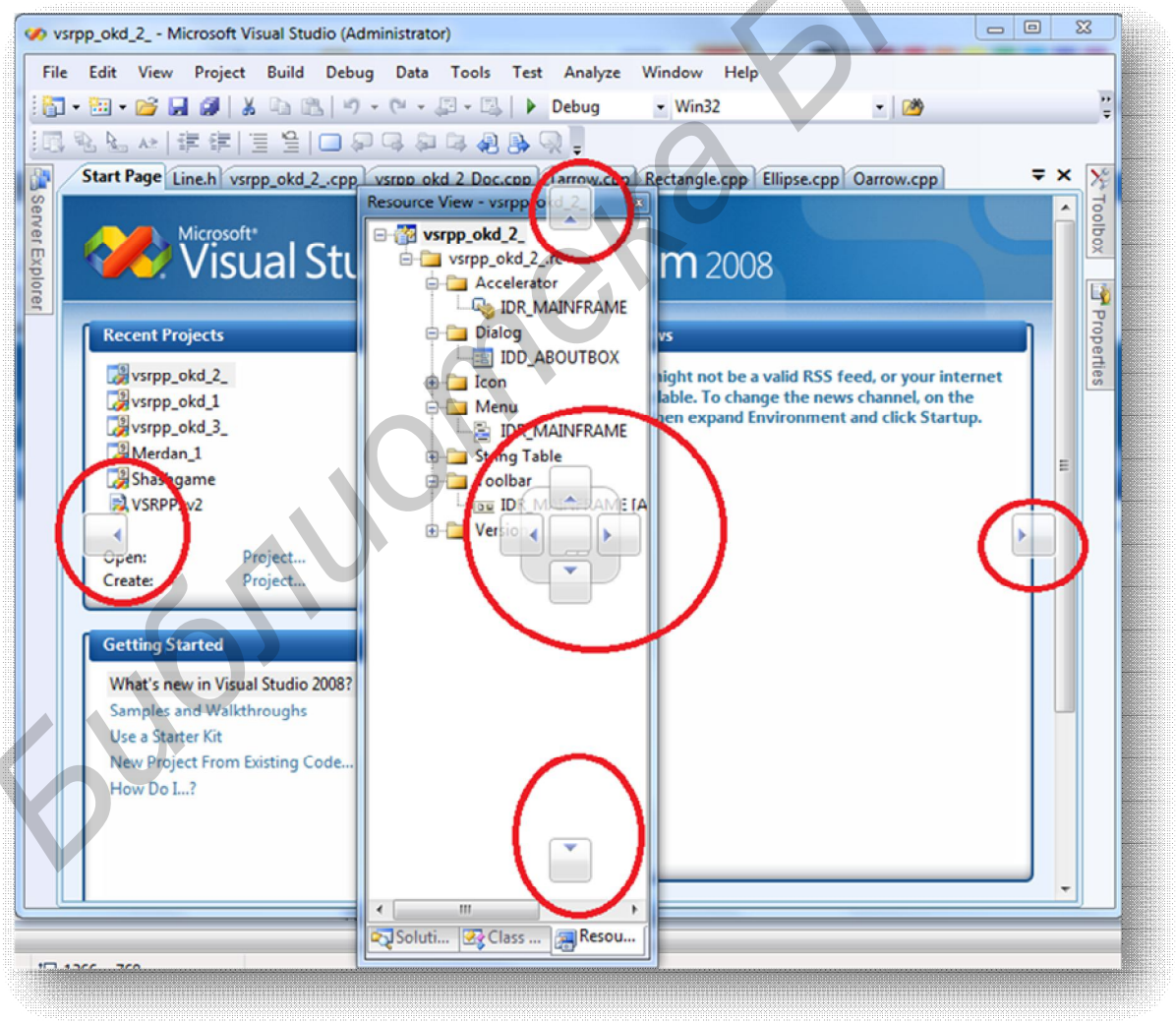

Рис. 1.1. Главное окно среды разработки

Recent projects – в этом разделе расположен список последних проектов, с которыми вы работали. При первом запуске этот список будет пуст.

Внизу списка есть две ссылки для открытия существующего проекта и создания нового проекта.

Get started – в этом разделе находятся ссылки на разделы файла помощи с информацией для тех, кто впервые работает с Visual Studio или впервые начинает разрабатывать программы.

Getting started – в этом разделе находятся ссылки на последние документы, связанные со средой разработки и программированием.

MSDN: Visual Headlines – занимает основную часть окна и содержит последние новости из мира языка программирования.

По краям окна могут находиться панели и окна, которые можно перемещать и располагать по своему усмотрению. В этих окнах находятся дополнительные инструменты, которые можно использовать во время работы. В данном примере эти окна свернуты, поэтому вы можете видеть на рис. 1.1 только тоненькие панели слева и справа. На этих панелях находятся заголовки панелей, которые будут всплывать, если навести на них курсором мыши. У каждой такой панели в заголовке есть название панели и три кнопки:

– эта кнопка вызывает меню настройки панели;

– если эта кнопка нажата, то панель будет закреплена на поверхности главного окна. Если кнопка отжата, то панель будет автоматически прятаться;

– закрыть текущую панель. Если вы закрыли какое-то окно, то впоследствии его можно будет открыть, выбрав название окна в меню View.

Для создания проекта следует выбрать File → New → Project. Окно создания нового проекта представлено на рис. 1.2. С левой стороны расположены различные категории проектов, а справа – иконки проектов, которые можно создавать. Внизу окна следует указать имя для нового приложения и путь, где оно должно располагаться.

1.1. Работа с решениями (Solution)

Панель решения Solution Explorer (эксплорер решения), представленную на рис. 1.3, можно включить, выбрав меню View → Solution Explorer.

Все проекты, которые создаются в Visual Studio, заключаются в решение. Одно решение – это как папка для проектов, которая может содержать несколько проектов. Это очень удобно, потому что очень часто решения действительно должны состоять из нескольких проектов, и благодаря объединению в одном виртуальном пространстве можно работать со всеми этими проектами как с одним целым.

Содержимое окна Solution Explorer представлено в виде дерева. Корнем дерева выступает имя решения. По умолчанию решения получают такое же имя, как и проект. Чтобы переименовать решение, нужно установить курсор на имени решения, вызвать контекстное меню и выбрать пункт Rename.

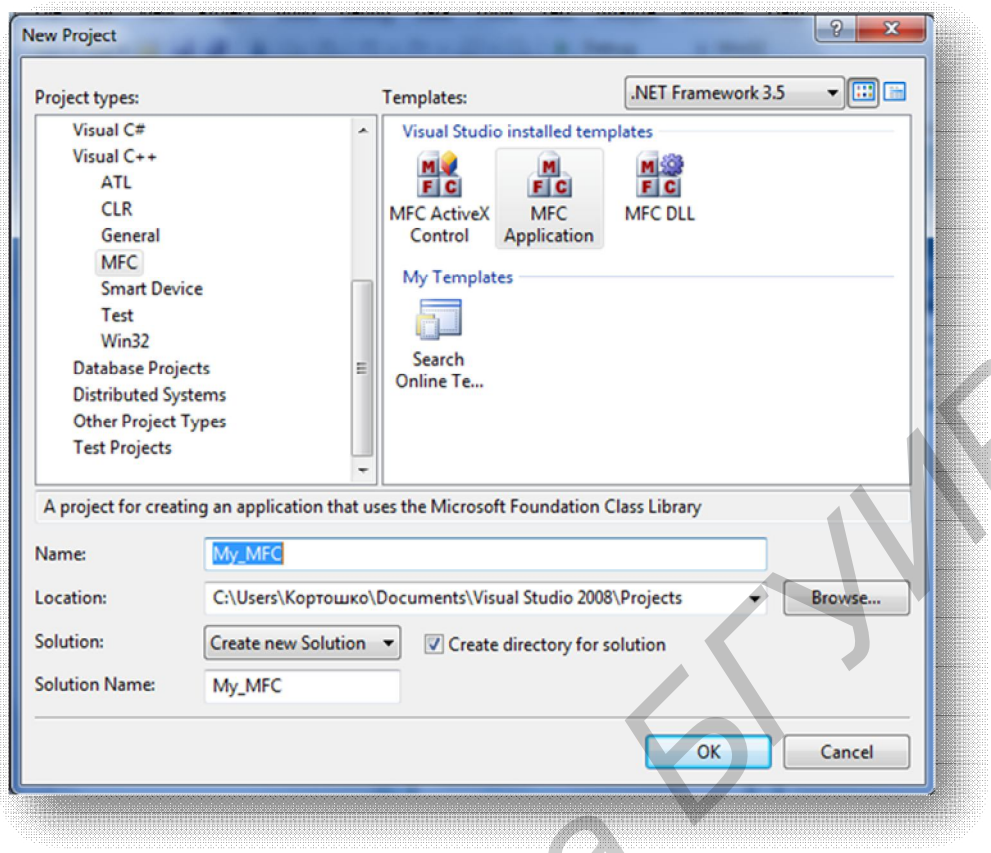

Рис. 1.2. Окно создания нового проекта

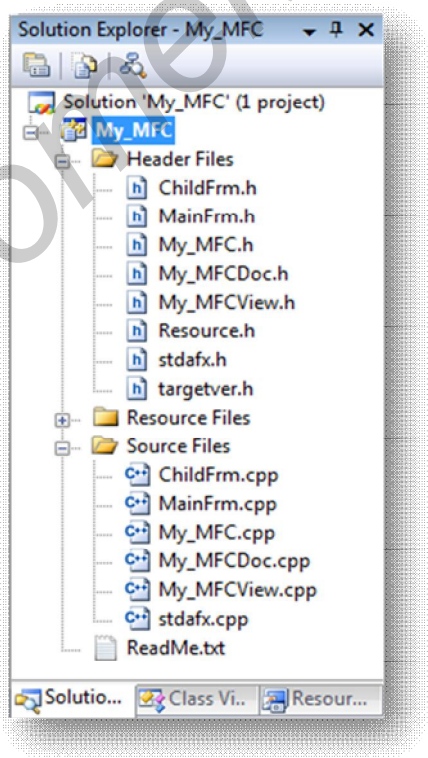

Рис. 1.3. Панель решения Solution Explorer

На месте имени решения появится встроенное поле для редактирования, в котором вы можете ввести новое имя для решения. Ввод имени нужно завершить нажатием клавиши Enter. Конфигурация решения сохраняется в файле с расширением .sln. Впоследствии, если вы захотите открыть все решение в целом, нужно открывать именно этот файл, а не файлы проектов, такие как csproj. Если в решении только один проект, то можно открывать файл проекта, решение будет открыто автоматически. Команды, которые есть в контекстном меню решения, представлены на рис. 1.4.

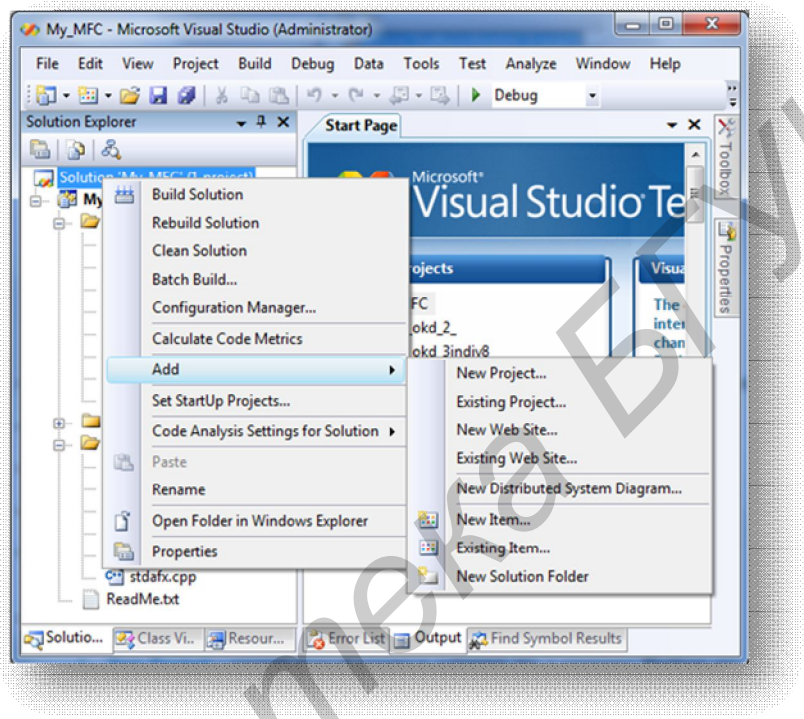

Рис. 1.4. Команды контекстного меню решения

Содержание контекстного меню решения следующее:

– Build Solution – собрать решение. Будут полностью собраны все проекты решения;

– Rebuild Solution – пересобрать решение;

– Clean Solution – очистить решение;

– Batch build – настройка проектов, которые нужно будет пересобрать;

– Configuration Manager – вызывает окно, в котором можно управлять конфигурациями решения;

– Add – этот пункт содержит подпункты, с помощью которых в решение можно добавить дополнительные проекты. Это могут быть:

1) New Project – создать новый проект и добавить его в решение. Выбрав это меню, вы увидите окно создания нового проекта, как при выборе меню File → New → Project;

2) Existing Project – отобразит стандартное окно открытия файла, с помощью которого вы можете найти уже существующий проект в вашей файловой системе и добавить его в решение;

3) New Web Site – позволяет создать проект нового WEB-сайта;

4) Existing Web Site – добавить в решение существующий сайт;

5) New Item – добавить в решение новый элемент;

6) Existing Item – добавить в решение существующий элемент;

7) New Solution Folder – создать новую папку в решении;

– Set StartUp Projects – вызывает окно конфигурации решения, в котором вы можете указать, какой из проектов и как должен запускаться при нажатии клавиши F5 во время разработки;

– Paste – вставить из буфера обмена;

– Rename – переименовать решение;

– Open Folder in Windows Explorer – открыть папку решения в окне проводника Windows;

– Properties – отобразить окно свойств решения.

Большинство из рассмотренных команд не требуют дополнительных пояснений. Рассмотрим меню Batch Build (окно настройки сборки), которое представлено на рис. 1.5. Здесь перечислены все проекты решения и указаны возможные конфигурации (вторая колонка списка).

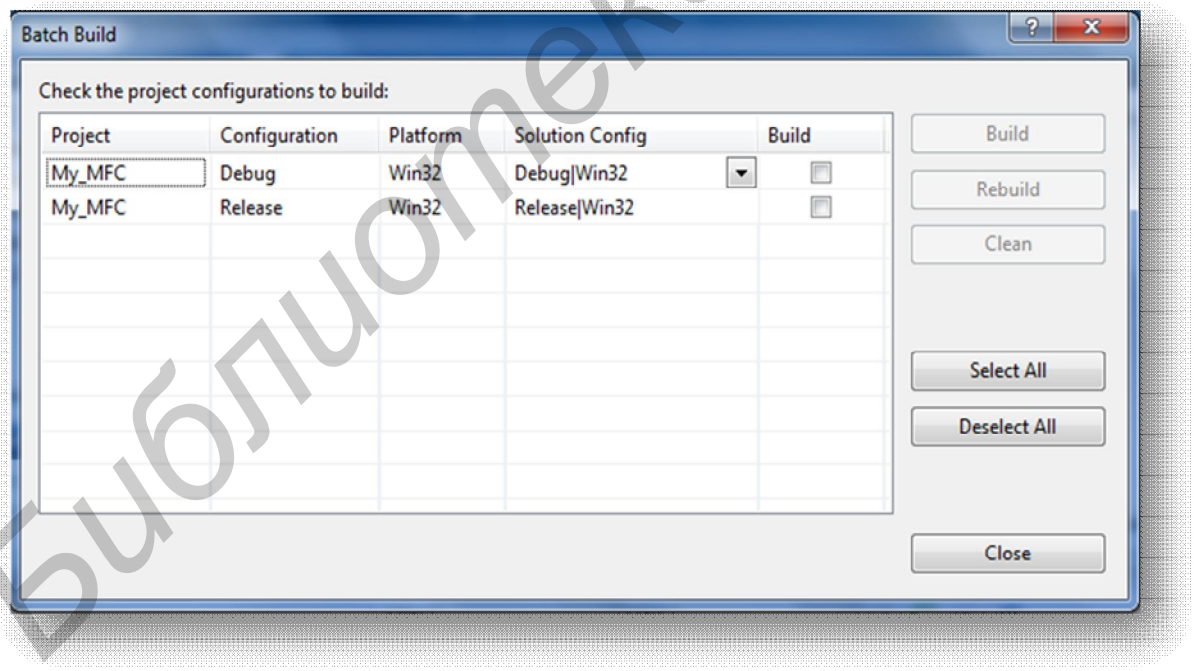

Рис. 1.5. Окно настройки сборки

В последней колонке Build можно указать, какие конфигурации и проекты вас интересуют. Нажав кнопку Build, или Rebuild, или Clean в этом окне, вы можете запустить процесс сборки, пересборки или очистки всех выбранных проектов.

Рассмотрим окно свойств решения, представленное на рис. 1.6. Оно отображается на экране, если выбрать в контекстном меню решения пункт Properties. Справа наверху находится выпадающий список, в котором можно настроить конфигурации решения. По умолчанию существует две конфигурации: Debug и Release. Команда Startup Project помогает выбрать проект, который должен запускаться при нажатии клавиши F5 во время разработки.

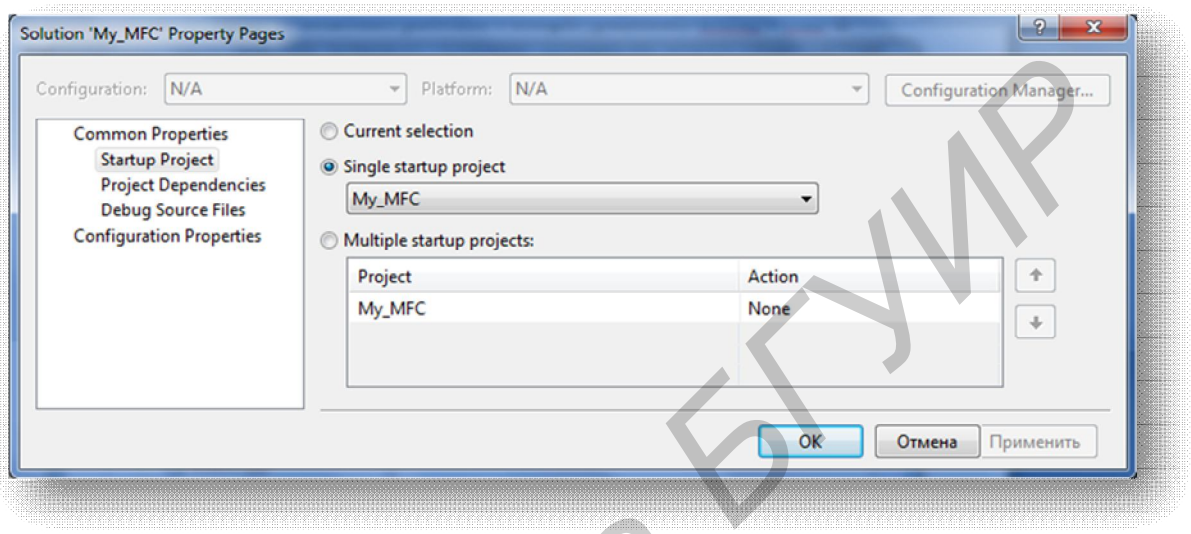

Рис. 1.6. Окно настройки свойств решения

В разделе Project Dependencies можно указать, от чего зависит проект. Например, исполняемый файл может зависеть от библиотеки кода. Это значит, что библиотека кода должна компилироваться раньше, чем исполняемый файл.

1.2. Работа с классами (Class View)

Панель Class View (представление классов) можно включить, выбрав меню View → Class View. В панели Class View, представленной на рис. 1.7, отображаются созданные классы, глобальные переменные и функции, макросы и константы, карты сообщений. В верхней части этого окна отображаются классы, а в нижней – элементы, входящие в выбранный класс. Контекстное меню для класса CMy_MFCDoc представлено на рис. 1.8.

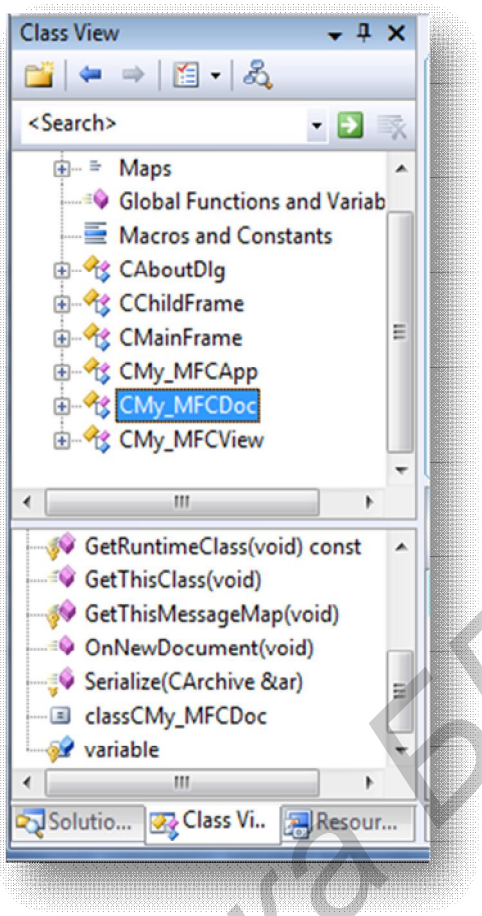

Рис. 1.7. Окно Class View

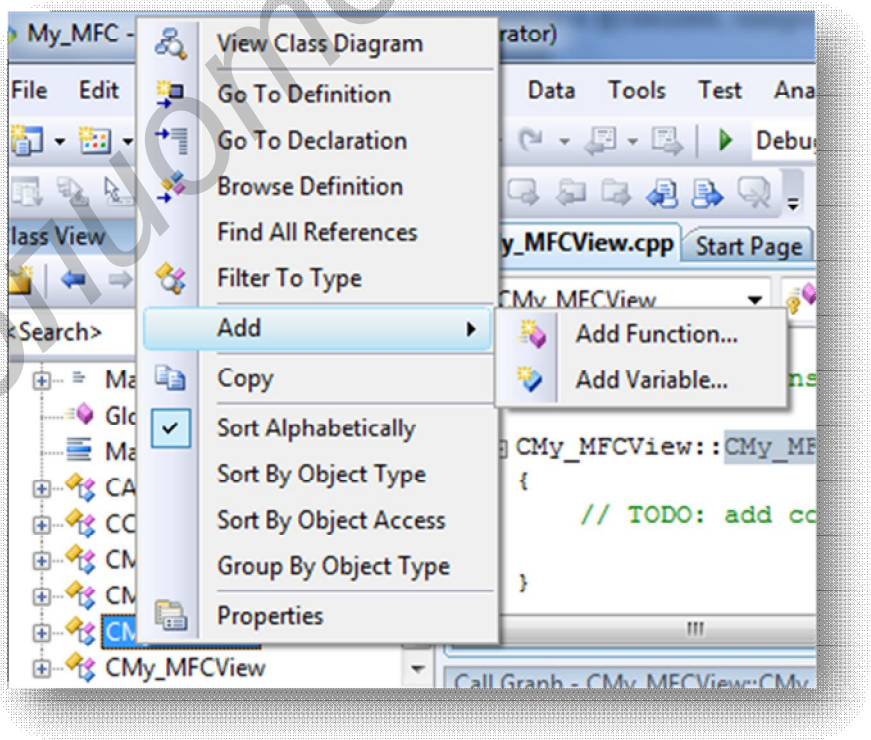

Рис. 1.8. Контекстное меню для выбранного класса

Контекстное меню содержит следующие пункты:

– View Class Diagram – просмотр (создание) диаграммы классов UML;
– Go To Definition – показать код описания (.cpp файл);
– Go To Declaration – показать код объявления (.h файл);
– Browse Definition – подключить определение;
– Find All References – найти все ссылки на данный класс;
– Filter To Type – показать только данный класс в данном окне (при помощи стрелок на панели можно вернуться);
– Add – содержит выпадающее меню:
– Add Function – добавить метод (будет вызван мастер);
– Add Variable – добавить переменную (будет вызван мастер);
– Сортировки и группировки для просмотра;
– Properties – вызовет окно свойств.

Рассмотрим окно свойств «Properties», представленное на рис. 1.9. Верхняя строка указывает на имя класса. Ниже (выделено) панель инструментов. Первые два элемента указывают вид просмотра: Categorized, Sort. Третий элемент Properties – просмотр свойств класса.

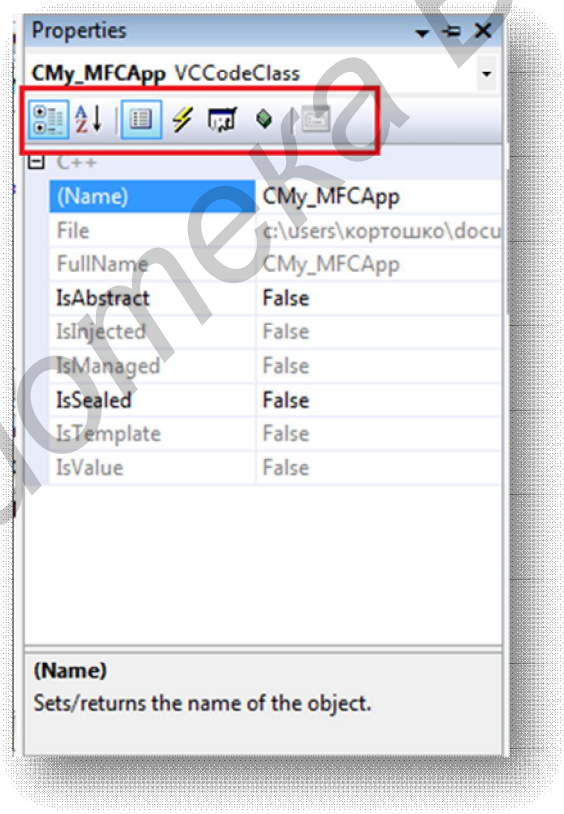

Рис. 1.9. Окно свойств для выбранного класса

На рис. 1.10 выделена закладка Events – просмотр командных сообщений, обрабатываемых данным классом. На рис. 1.11 выделена закладка Messages – просмотр обрабатываемых сообщений, поступающих окну. На рис. 1.12 выделена закладка Overrides – просмотр виртуальных функций – обработчиков стандартных действий.

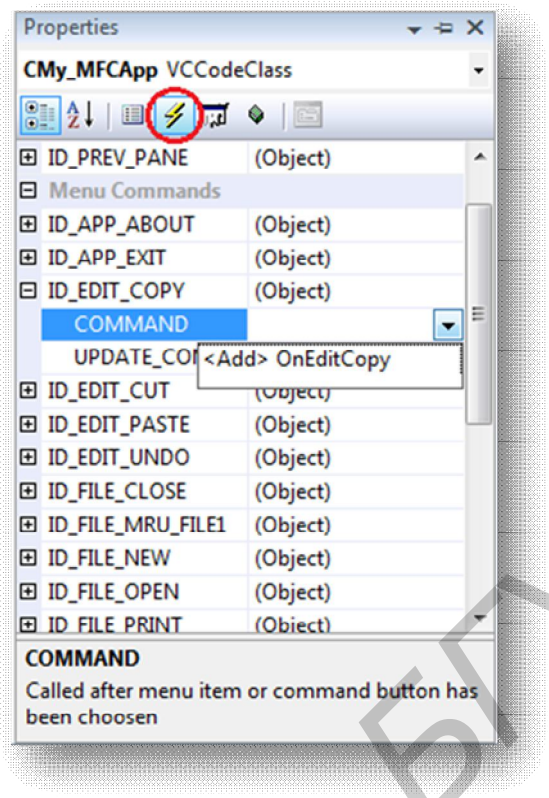

Рис. 1.10. Закладка Events

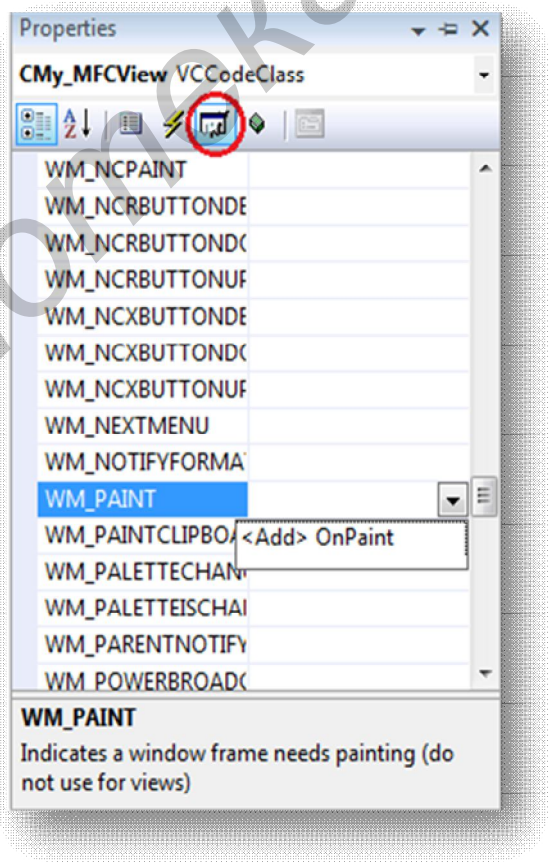

Рис. 1.11. Закладка Messages

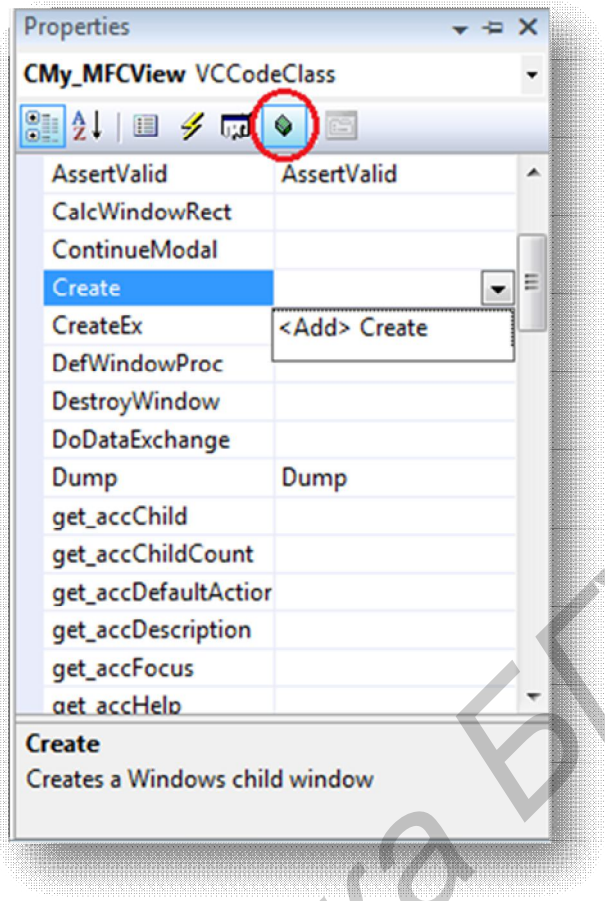

Рис. 1.12. Закладка Overrides

Во всех случаях в нижней части панели показывается поясняющее сообщение.

1.3. Работа с ресурсами (Resource View)

Панель Resource View показывает файлы – ресурсы, включенные в проект, как показано на рис. 1.13. Ее можно включить, выбрав меню View → Other Windows → Resource View. Все ресурсы, которые созданы в проекте, представлены в этой панели. В ней отображаются созданные акселераторы (обработчики нажатий клавиш), диалоговые окна, иконки приложения, меню, таблицы строк, панели инструментов, версия приложения.

При нажатии правой кнопки мыши в любом месте данной панели вызывается контекстное меню, как изображено на рис. 1.14. Команда Resource Includes вызовет окно, хранящее имя файла с основными макросами связывания ресурсов с данными, Resource Symbols – вызовет окно редактирования макросов связи, Add Resource – вызовет окно добавления ресурса. В нем можно выбрать вид ресурса, например диалоговое окно, и, вызвав контекстное меню, добавить копию выбранного ресурса или добавить ресурс, соответствующий данному.

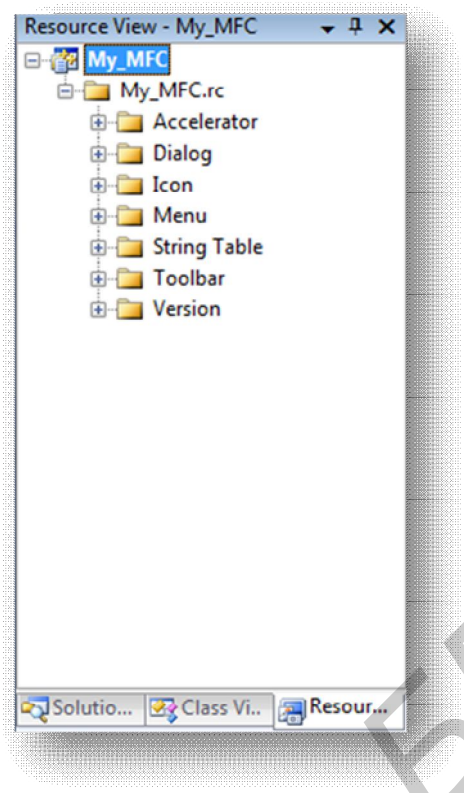

Рис. 1.13. Панель Resource View

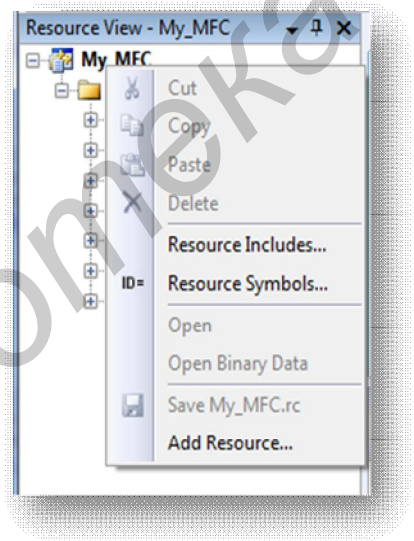

Рис. 1.14. Контекстное меню для панели Resource View

1.4. Окно редактирования текстов программ

В рамках интегрированной среды созданы удобные средства для редактирования программ в процессе их разработки. Здесь реализованы основные средства редактирования – те же, что и в известных системах, например Microsoft Office:

– вставить/скопировать/вырезать текст реализуется клавишами Ctrl + V/C/X;

– отмена/повторить реализуется клавишами Ctrl + Z/Y.

VS выделяет элементы программы с помощью синтаксической раскраски. По умолчанию текст – черный, комментарии – зеленые, ключевые и служебные слова – голубые, строковые значения – красные. Можно сформировать специальные расцветки для строковых переменных, чисел, используя вкладку Format в диалоговом окне Options. Для этого необходимо выбрать команду Tools → Options. Если щелкнуть правой кнопкой мыши на поле редактируемого файла, появляется контекстное меню, представленное на рис. 1.15.

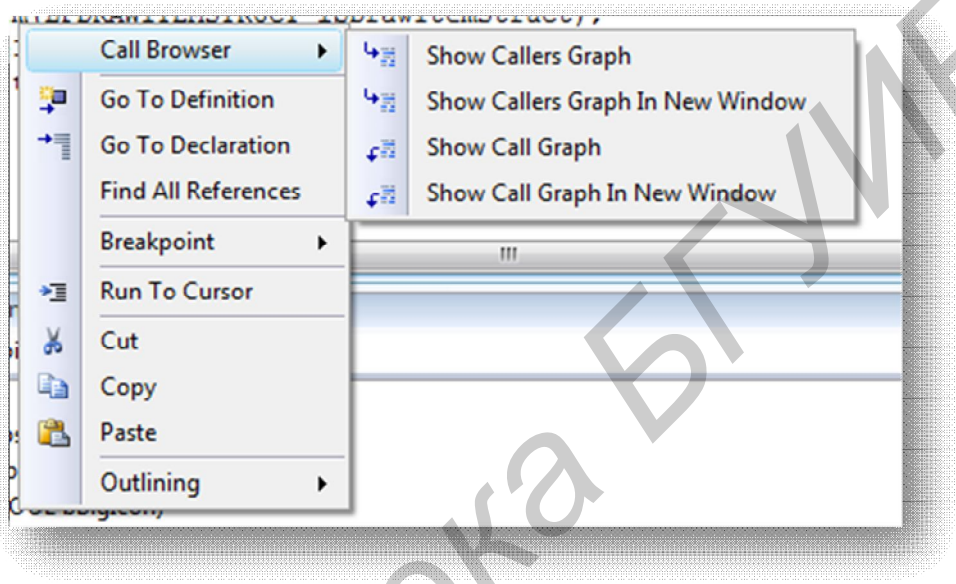

Рис. 1.15. Контекстное меню редактируемого файла

Значение команд, представленных в нем, следующее:

– Cut – удаляет выбранный текст в системный буфер Clipboard;

– Copy – копирует выбранный текст в системный буфер Clipboard;

– Paste – заменяет или вставляет;

– Go To Definition – открывает файл, определяющий элемент, на который указывает курсор (файл заголовка – для переменной, файл текста программы – для функции);

– Go To Declaration – открывает файл, объявляющий элемент, на который указывает курсор (файл заголовка);

– Find All References – находит все строки в файлах проекта, где встречается выбранное слово;

– Call Browser – браузер вызовов выбранной функции;

– Breakpoint – добавление контрольных точек;

– Run To Cursor – выполняет программу до строки кода, помеченной курсором.

1.5. Система меню

Существует два способа выбора команд из меню. Более распространенный из них состоит в том, что вы устанавливаете указатель мыши на нужных командах меню и щелкаете левой кнопкой мыши. Второй способ заключается в использовании клавиш быстрого вызова, которые выделяются подчеркиванием в названиях команд. Так, меню File можно раскрыть, нажав одновременно [Alt+F]. Существует еще один способ вызова отдельных команд в любой момент времени, а именно с помощью предварительно заданных «горячих» клавиш. Если для команды определено сочетание клавиш, то это сочетание будет указано в меню справа от соответствующего пункта, как показано в рис. 1.16. Например, в меню File есть команда New..., которую можно вызвать, просто нажав [Ctrl+N].

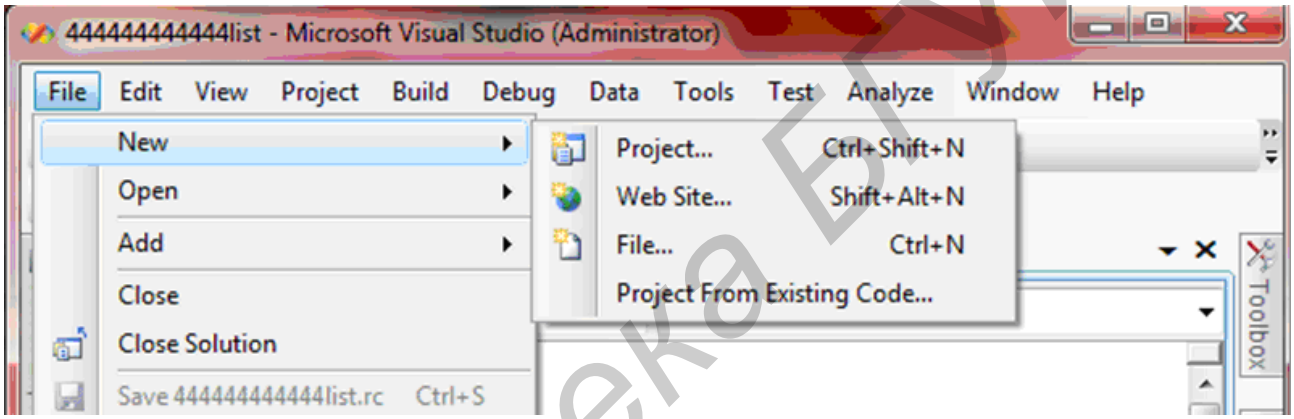

Рис. 1.16. Вызов команд меню

Команда меню, показанная серым цветом, является в данный момент недоступной — вероятно, отсутствуют некоторые условия, необходимые для ее выполнения. Например, команда Save из меню File будет недоступной, если в редактор ничего не загружено. Если за названием команды меню следует троеточие, значит, после выбора данной команды будет открыто диалоговое окно. Например, после выбора команды Open... в меню File открывается диалоговое окно Open.

Многие команды меню представлены кнопками на панелях инструментов, которые обычно размещаются непосредственно под строкой меню.

VS имеет много меню. Некоторые команды находятся на третьем или четвертом уровне иерархии. В большинстве случаев ту же задачу можно решить значительно быстрее. В строке VS есть 12 меню:

- File – создание/открытие/сохранение/закрытие проекта/файла;
- Edit – редактирование;
- View – меню управления видами и средствами VS;
- Project – меню команд управления, разрабатываемых проектом в целом;
- Build – компиляция, компоновка;
- Debug – отладка;

– Data – данные;
– Tools – настройки VS и доступ к автономным утилитам;
– Test – задание параметров тестирования;
– Analyze – анализ и сбор информации (отчеты);
– Window – выбор/изменение расположения/закрытие окон редактирования;
– Help – обращение к системе Справок.

Рассмотрим некоторые из них.

1.5.1. Меню File

Пункты меню File полностью включают традиционные операции над файлами (например принятые в системе Word) и дополнительные операции, учитывающие специфику VS. Некоторые пункты меню снабжены значками панели инструментов и горячими клавишами.

Выбор пункта File → New приводит к появлению таких пунктов, как создание нового проекта, веб-сайта, файла или нового проекта на основе уже существующего кода программы, как представлено на рис. 1.17. Пункт File → Open – открытие проектов / сайтов / файлов или их конвертация, как представлено на рис. 1.18. Пункт File → Add – добавление новых/существующих проектов, веб-сайтов и дистрибутированных системных диаграмм к используемому решению (solution) (рис. 1.19).

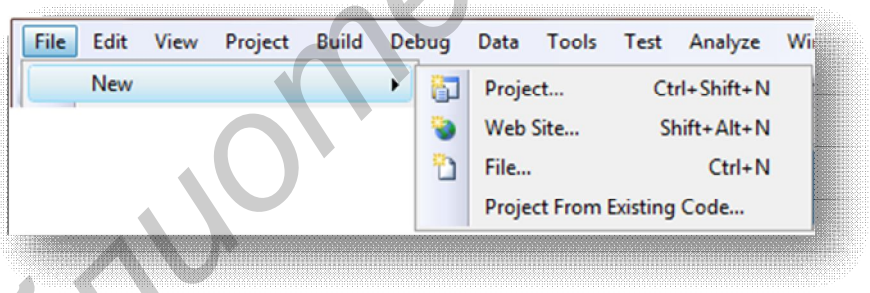

Рис. 1.17. Пункт меню File → New

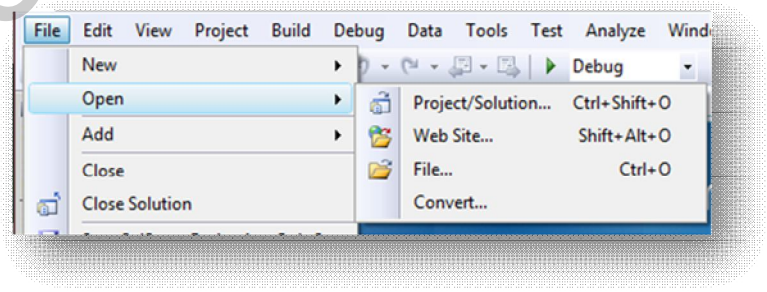

Рис. 1.18. Пункт меню File → Open

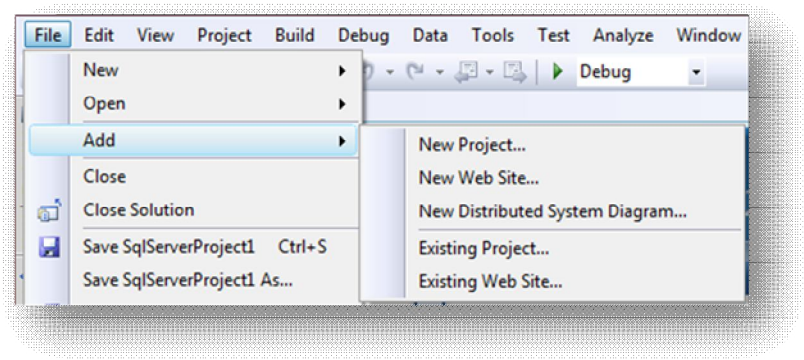

Рис. 1.19. Пункт меню File → Add

Команда Close предназначена для закрытия ранее открытого файла. Если у вас в настоящий момент открыто несколько файлов, данная команда закроет активное, т. е. текущее окно. Команда Close Solution закроет открытое в настоящий момент рещение. Команда Save сохраняет содержимое текущего окна в соответствующем файле. Для сохранения файла можно также использовать расположенную на панели инструментов кнопку Save (третья слева). Если файл был открыт в режиме только для чтения, то команда Save будет недоступной. Команда SaveAs... позволяет сохранить содержимое окна в файле под новым именем. Команда Save All сохраняет все файлы, также все файлы сохраняются непосредственно перед компиляцией и когда закрывается приложение. Advanced Save Options – другие варианты сохранения. Page Setup – настройка печати.

Пункты меню Recent Files и Recent Projects содержат имена файлов и рабочей среды, которые были открыты последними.

1.5.2. Меню Edit

Пример меню Edit представлен на рис. 1.20. Команды Undo и Redo позволяют отменить (или восстановить) последнюю выполненную операцию редактирования. Команда Cut копирует выделенный блок текста из активного окна в буфер обмена, после чего удаляет этот блок из окна. Команду Cut обычно используют в сочетании с командой Paste для перемещения блока текста из одного места в другое.

Команды Copy, Paste, Delete, SelectAll, Find and Replace работают так же, как и в большинстве текстовых редакторов.

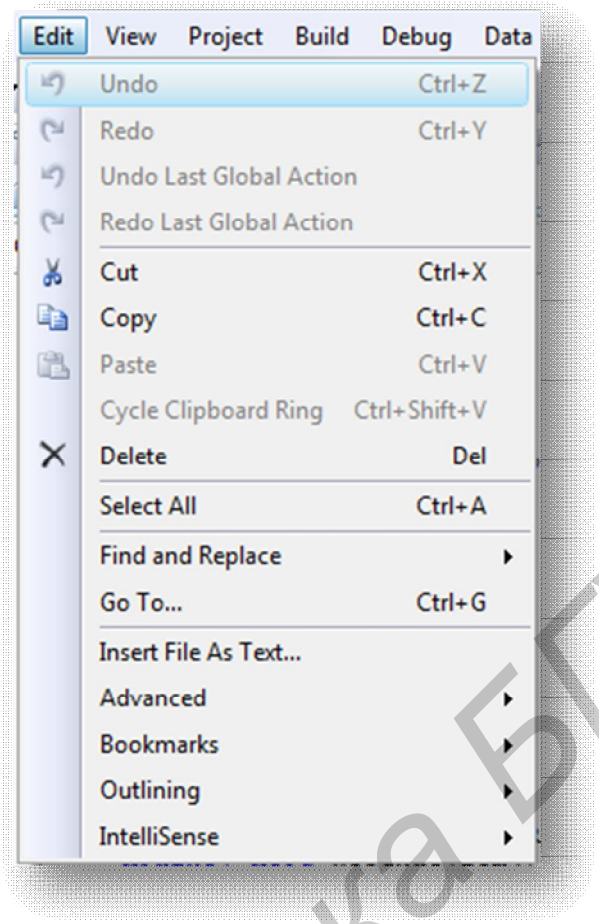

Рис. 1.20. Меню Edit

С помощью команды GoTo... можно быстро переместить курсор к разным частям проекта (адрес, закладка, определение, ошибки, строка, смещение (offset), ссылка к объявлению функции или идентификатору).

Команда Advanced предназначена для быстрого изменения текста. Команда Bookmarks... позволяет помещать закладки в тех местах программы, к которым программист часто обращается. После того как закладка будет установлена, можно быстро перейти к ней с помощью команды меню или определенного сочетания клавиш. Закладку, которая больше не понадобится, можно в любой момент удалить. Можно создавать как именованные (они будут сохраняться между сеансами редактирования), так и безымянные закладки. К именованной закладке можно перейти в любое время, даже если файл, к которому она относится, в данный момент не открыт. Именованная закладка хранит как номер строки, так и позицию курсора на строке, которую он занимал во время ее создания. Причем позиция будет автоматически обновляться по мере редактирования файла.

Команда Outlining позволяет группировать строки кода. Команда IntelliSense позволяет получить список членов класса/пространства имен, информацию о параметрах функций, быструю информацию об элементе без поиска места их определения в файле, закончить начатое слово, как представлено на рис. 1.21.

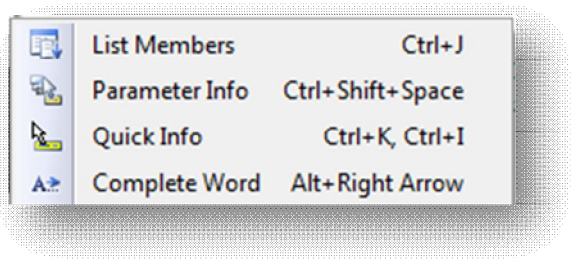

Рис. 1.21. Меню Outlining

Команда List Members отображает список доступных переменных-членов или функций выбранного класса либо структуры, активизирует функцию Autocomplete. Команда QuickInfo отображает окно подсказки с типом переменной, на которой стоит курсор. Команда ParameterInfo отображает полное описание (включая список параметров) функции, имя которой расположено слева от курсора. Параметр, выделенный полужирным шрифтом, соответствует тому параметру, который вы должны ввести в данный момент. При выборе команды CompleteWord программа автоматически допишет вместо вас название функции или имя переменной, которое вы только начали вводить.

1.5.3. Меню View

Меню View, представленное на рис. 1.22, содержит команды, позволяющие настроить внешний вид рабочего пространства. Пункты меню позволяют открыть следующие окна:

– Code – код выбранного элемента;
– Open – выбранный файл;
– Open With… – диалоговое окно, с помощью которого можно выбрать программу для открытия выбранного файла;
– Server Explorer – окно для соединения с сервером;
– Solution Explorer – окно решений;
– Bookmark Window – окно, содержащее список закладок;
– Class View – окно представления классов;
– Code Definition Window – окно, содержащее код определения макроса;
– Object Browser – браузер объектов;
– Error List – окно списка ошибок;
– Output – окно выходного потока;
– Properties Window – окно свойств;
– Task List – окно задач;
– Toolbox – окно инструментов для приложения;
– Find Results – окно результатов поиска;
– Other Windows – другие вспомогательные окна, среди которых окно вызовов, командная строка, менеджер свойств, окно ресурсов и т. д.;

– Toolbars – содержит список панелей инструментов для рабочего пространства, которые вы можете сами выбрать;

– Full Screen – работа со средой в режиме во весь экран;

– Navigate Backward – навигация обратно;

– Navigate Forward – навигация вперед;

– Property Pages – страницы свойств.

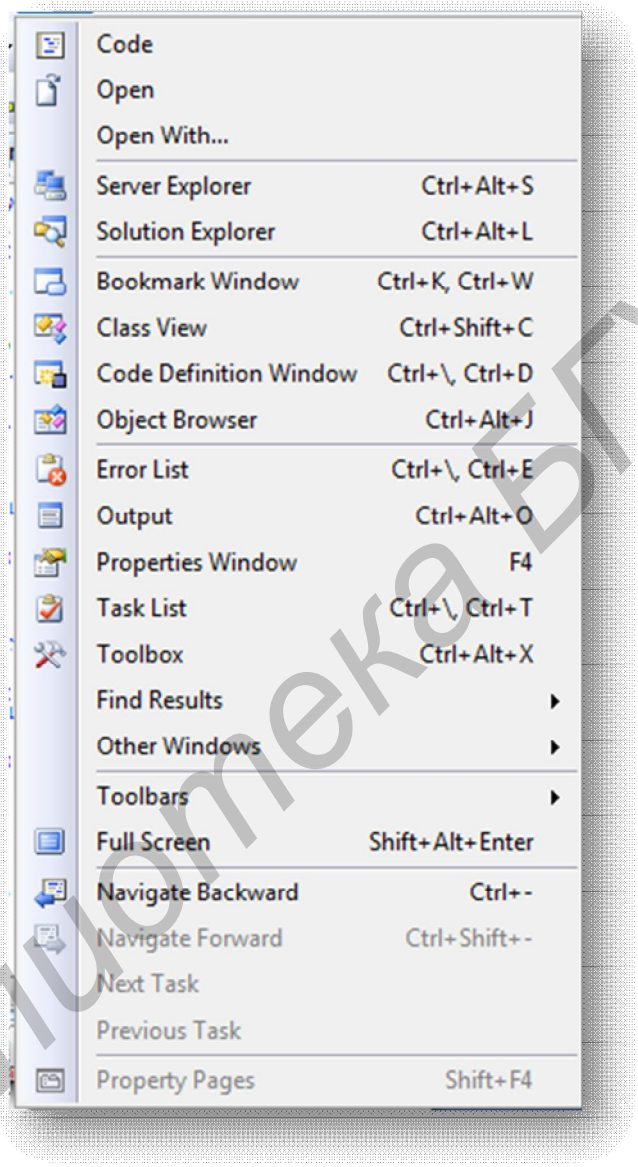

Рис. 1.22. Меню View

1.5.4. Меню Project

Меню Project, представленное на рис. 1.23, содержит пункты, связанные с сопровождением проекта. Рассмотрим некоторые из них:

– Add Class… – вызывает диалоговое окно добавления к проекту класса;

– Add New Item… – окно добавления к проекту нового элемента (файла/документа);

– Add Existing Item… – окно добавления к проекту уже существующего элемента (файла/документа);

– Show All Files – показать все файлы проекта;

– Set as StartUp Project – установить как проект начала;

– Refresh Project Toolbox Items – обновить элементы проекта;

– Properties… – вызовет окно свойств проекта. При выборе этой команды открывается диалоговое окно, позволяющее устанавливать практически все параметры конфигурации проекта, включая опции компилятора и компоновщика.

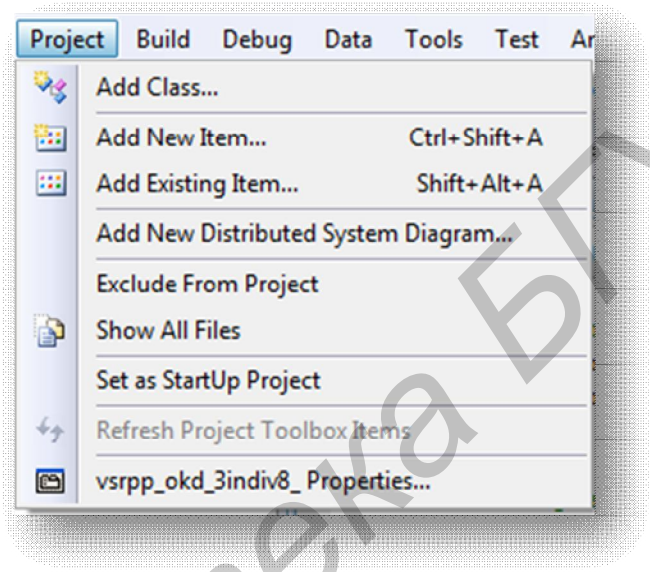

Рис. 1.23. Меню Project

1.5.5. Меню Build

Меню Build, представленное на рис. 1.24, включает команды, связанные с компиляцией, выполнением и отладкой приложений. Рассмотрим некоторые пункты:

– Compile – выбор этой команды приводит к компиляции содержимого текущего окна;

– Biuld Solution/Project – выполняется автоматический анализ файлов проекта, компилируются только те из них, которые были созданы позже, чем исполняемый файл проекта;

– Rebuild Solution/Project – компилируются все файлы проекта, не обращая внимания на дату и время создания файлов. Если при выполнении команды будут обнаружены синтаксические ошибки, как фатальные, так и потенциально опасные, то предупреждения и сообщения о них появятся в окне Output;

– BatchBuild... – команда аналогична команде Build, но с ее помощью можно обработать сразу несколько конфигураций одного проекта;

– Clean Solution/Project – из всех конфигураций текущего проекта удаляются промежуточные файлы. Построить файлы заново можно путем выбора команды Build;

– Project Only – задачи только для данного проекта;

– Profile Guided Optimization – опции для оптимизации проекта;

– Configuration Manager – окно конфигураций компиляции.

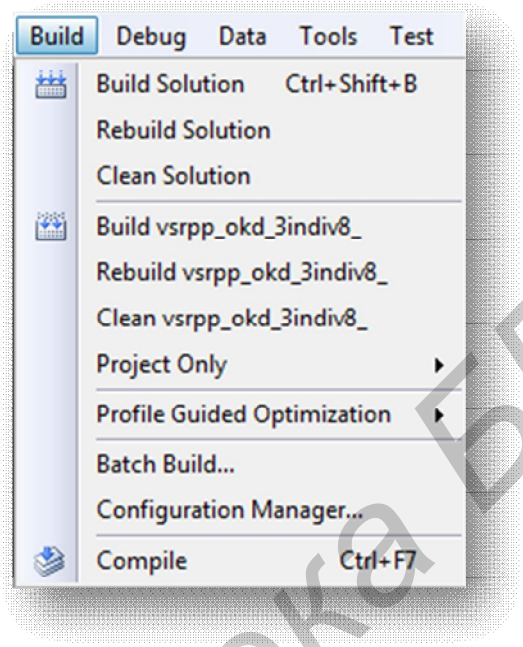

Рис. 1.24. Меню Build

1.5.6. Меню Debug

Данное меню, представленное на рис. 1.25, содержит команды, предназначенные для выполнения программы в режиме отладки: до курсора или до заданной точки останова. Основные пункты следующие:

– Windows – выбор необходимых окон при отладке;

– Start Debugging – начать отладку;

– Start With Application Verifier – начать с верификатором приложения;

– Start Without Debugging – начать без отладки;

– Attach to Process… – подключить к процессу;

– Exceptions… – окно установки прерывания при возникновении исключения;

– Step Into – переход в метод(функцию) при отладке;

– Step Over – переход к следующему действию при отладке;

– Toggle Breakpoint – переключает точку останова;

– New Breakpoint – создает новую точку останова;

– Delete All Breakpoints – удаляет все точки останова.

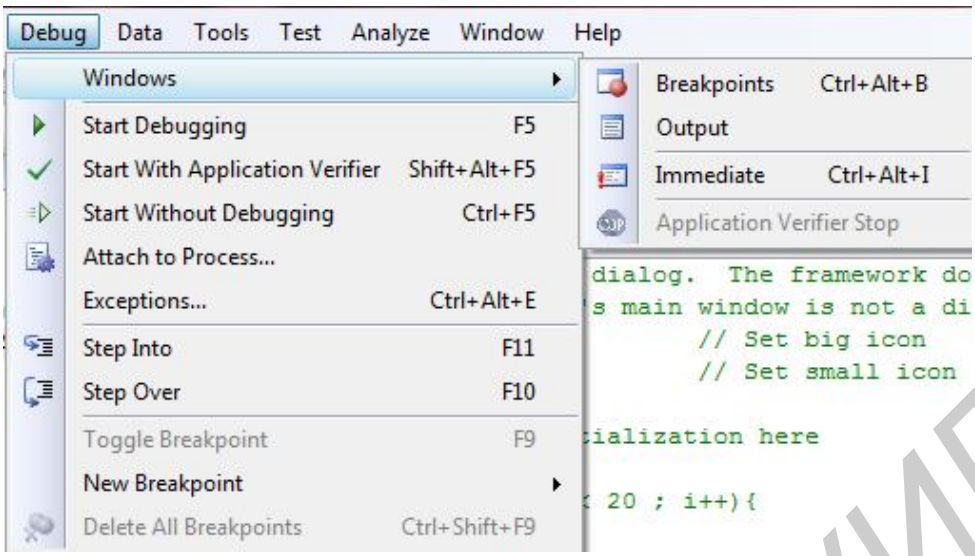

Рис. 1.25. Меню Debug

1.5.7. Меню Tools

Меню Tools позволяет управлять использованием инструментальных средств. Дополнительные средства можно добавить с помощью пункта Customize, который позволяет добавлять новые пиктограммы в панель инструментов, управлять отображением панели инструментов, добавлять пункты в меню Tools и т. д.

Назначение пунктов меню следующее:

– ErrorLookup – используют при необходимости получить текст сообщений, связанных с кодами системных ошибок. Если ввести код ошибки в поле Value, в поле ErrorMessage автоматически отобразится связанное с ним сообщение;

– Spy++ – выводит информацию о выполняющихся системных процессах и потоках, существующих окнах и поступающих оконных сообщениях. Указанная утилита также предоставляет набор инструментов, облегчающих поиск нужных процессов, потоков и окон;

– ATL / MFC Trace – дает дополнительные возможности для отладки оконных приложений, построенных на основе MFC. Эта утилита отображает в окне отладки сообщения о выполнении операций, связанных с использованием библиотеки MFC, а также предупреждения об ошибках, если при выполнении приложения происходят какие-либо сбои;

– Customize... – открывает диалоговое окно Customize, которое позволяет настраивать меню и панели инструментов, а также назначать различным командам сочетания клавиш;

– Options... – открывает окно Options, в котором задаются различные параметры среды Visual C++.

1.5.8. Меню Help

В VS справочная система является отдельным программным продуктом. В большинстве диалоговых окон нажатие клавиши F1 вызывает стандартную справочную систему, так же как и меню Help.

Основные пункты этого меню следующие:

– Search – поисковик в справочнике;

– Contents – позволяет запустить программный продукт MSDN, если последний не был запущен ранее, и вывести на экран вкладку Table of Contents;

– Index – позволяет открыть вкладку Index продукта MSDN;

– Technical Support – отображает раздел справочной системы, посвященный вопросам технической поддержки;

– About Mircosoft Visual Studio – идентификатор продукта, отображает информацию о версии программы, авторских правах, зарегистрированном пользователе и установленных дополнительных компонентах.

Доступ к справочной системе VisualC++ облегчен благодаря тому, что вся информация предоставляется в интерактивном режиме. Чтобы получить справку, достаточно навести указатель на интересующий вас инструмент и нажать клавишу [F1]. Если навести указатель на элемент программного кода C/C++ и нажать [F1], то отобразится справка о синтаксисе выбранной конструкции.

2. Принципы функционирования программ под Windows

2.1. Концепции и средства программирования в Windows

2.1.1. Среда Windows

Windows, как известно, представляет собой графическую многозадачную операционную систему. Все программы, разработанные для этой среды, должны соответствовать определенным стандартам и требованиям [5]. Это касается прежде всего внешнего вида окна программы и принципов взаимодействия с пользователями. Благодаря стандартам, общим для всех приложений Windows, пользователю не составляет труда разобраться в принципах работы любого приложения.

Чтобы помочь программистам в разработке приложений для Windows, были созданы многочисленные системные функции, позволяющие легко добавлять в создаваемые программы контекстные меню, полосы прокрутки, диалоговые окна, значки и многие другие элементы пользовательского интерфейса.

Windows позволяет работать с различными периферийными устройствами, такими как монитор, клавиатура, мышь, принтер и т. д., вне зависимости от типа самих устройств. Это дает возможность запускать одни и те же приложения на компьютерах с разной аппаратной конфигурацией.

2.1.2. Графический интерфейс пользователя

Первое, что бросается в глаза при знакомстве с приложениями Windows, – это стандартизированный графический интерфейс. Для представления дисков, файлов, папок и других системных объектов используются специальные растровые изображения, называемые значками. Окно типичного приложения Windows содержит, как правило, строку заголовка окна, меню, панель инструментов, строку состояния и рабочую область окна.

Название приложения появляется в строке заголовка его окна. Все функции, выполняемые программой, перечислены в меню. Чтобы выполнить команду, достаточно выбрать в меню соответствующий пункт. В большинстве приложений вызовы команд меню дублируются также комбинациями клавиш.

Как правило, программы имеют сходный внешний вид окон и похожие наборы команд меню. Пользователю достаточно на примере одного приложения изучить основные операции по открытию и манипулированию файлами и данными, чтобы чувствовать себя вполне уверенно с любым другим приложением Windows.

Стандартный интерфейс облегчает работу не только пользователям, но и программистам. Для добавления в приложение меню или диалогового окна достаточно воспользоваться одной из стандартных функций Windows. По-

скольку за реализацию меню и диалоговых окон отвечает сама система, а не программист, это гарантирует правильность работы интерфейса приложения.

2.1.3. Ввод данных посредством очередей

В среде Windows память компьютера представляет собой совместно используемый ресурс. Таковыми являются и большинство устройств ввода, в частности клавиатура и мышь. В среде Windows приложение не может обращаться напрямую к клавиатуре или мыши и получать данные непосредственно от них. Подобная задача выполняется самой Windows, которая заносит данные в системную очередь. Из очереди введенные данные распределяются между запущенными программами. Это осуществляется путем копирования сообщений из системной очереди в очереди соответствующих приложений. Затем, как только приложение оказывается готовым принять данные, оно считывает сообщения из своей очереди и распределяет их между открытыми окнами.

В Windows данные от устройств ввода поступают в виде сообщений. В каждом сообщении указывается системное время, состояние клавиатуры, код нажатой клавиши, позиция указателя мыши и состояние кнопок мыши, а также идентификатор устройства, пославшего сообщение.

Все сообщения от клавиатуры, мыши и системного таймера имеют одинаковый формат и обрабатываются схожим образом. В случае сообщений клавиатуры передается виртуальный код нажатой клавиши, который идентифицирует клавишу независимо от имеющейся в наличии клавиатуры, и генерируется Windows, а также аппаратно-зависимый скан-код, генерируемый самой клавиатурой. Передается также информация о состоянии ряда других клавиш, таких как [NumLock], [Alt], [Shift] и [Ctrl].

Клавиатура и мышь являются совместно используемыми ресурсами. Они посылают сообщения всем приложениям, открытым в данный момент в среде Windows. Система переадресовывает все сообщения клавиатуры текущему активному окну. Сообщения мыши обрабатываются иначе. Они направляются тому приложению, над чьим окном в данный момент находится указатель мыши. О таком окне говорят, что оно получает фокус.

Другую группу составляют сообщения системного таймера. Формат их такой же, как у сообщений клавиатуры и мыши. Windows позволяет приложениям инициализировать таймер таким образом, чтобы одно из окон приложения регулярно принимало сообщения от таймера. Такие сообщения поступают непосредственно в очередь данного приложения.

2.1.4. Сообщения

В основе Windows лежит механизм генерации и обработки сообщений. С точки зрения приложений сообщения являются формой уведомления о произошедших событиях, на которые приложение должно (или не должно) каким-то образом реагировать. Пользователь может инициировать событие

щелчком или перемещением указателя мыши, изменением размера окна или выбором команды из меню. Другие события могут быть инициализированы самим приложением. Сообщения могут генерироваться автоматически самой Windows, например в случае завершения работы системы, когда каждому открытому приложению посылается уведомление о необходимости проверить сохранность данных и закрыть свои окна.

Рассматривая роль сообщений в Windows, необходимо отметить, что именно благодаря подсистеме сообщений становится возможной многозадачность Windows. Подсистема сообщений позволяет ей распределять время работы процессора между всеми открытыми приложениями. Каждый раз, когда Windows посылает сообщение приложению, она на некоторое время открывает этому приложению доступ к процессору. Единственная возможность для приложения получить доступ к процессору – это получить сообщение от Windows. Кроме того, сообщения позволяют приложению прореагировать определенным образом на событие, произошедшее в системе. Это событие могло быть вызвано самим приложением, другим приложением, выполняющимся в это же время в Windows, пользователем или операционной системой. Каждый раз, когда происходит то или иное событие, Windows оповещает о нем все заинтересованные приложения, открытые в настоящий момент.

2.1.5. Аппаратная независимость

Windows обеспечивает также независимость пользовательской среды от типа используемых аппаратных устройств. Благодаря этому программисты избавляются от необходимости заранее учитывать, какие именно монитор, принтер или устройство ввода будут подключены к конкретному компьютеру.

В среде Windows драйверы для определенного устройства создаются и инсталлируются только один раз. Причем они могут устанавливаться сразу при инсталляции Windows, поступать вместе с программными продуктами или добавляться самим пользователем.

Аппаратная независимость приложений в Windows реализуется за счет того, что приложения не обмениваются данными с устройствами напрямую, а используют в качестве посредника Windows. Для приложения нет необходимости знать, какой именно принтер подключен к данному компьютеру. Программа передает команду Windows напечатать прямоугольную область с заливкой, после чего уже операционная система разбирается, как реализовать эту задачу на имеющемся оборудовании. Аппаратная независимость облегчает жизнь не только программистам, но и пользователям, поскольку избавляет их от необходимости выяснять, с какими типами внешних устройств может работать то или иное приложение.

Аппаратная независимость достигается также за счет определения минимального набора базовых операций, которые должны выполняться любыми устройствами. Этот минимальный набор включает элементарные операции, не-

обходимые для правильного решения любых задач, которые могут быть поставлены перед данным устройством. Какой бы сложности ни была задача, она может быть сведена к последовательности элементарных операций, входящих в набор минимальных требований к устройству. Например, далеко не все графопостроители обладают функцией рисования окружностей. Тем не менее даже если выведение окружностей не входит в набор функций графопостроителя, вы все равно можете создавать программы, которые могут ставить перед графопостроителем подобную задачу. И поскольку, в соответствии с минимальными требованиями, все графопостроители, инсталлируемые в Windows, должны уметь рисовать линии, Windows автоматически преобразует окружность в набор линий и передаст массив векторов графопостроителю.

Чтобы избежать конфликтов при вводе данных, в Windows также определен минимальный набор кодов клавиш, которые должны распознаваться любым приложением. Стандартный набор кодов в целом соответствует раскладке ПК-совместимой клавиатуры. Если какая-нибудь фирма выпустит клавиатуру с дополнительными клавишами, не входящими в стандартный набор, то она должна позаботиться о специальном программном обеспечении, преобразующем коды этих клавиш в последовательности стандартных кодов, распознаваемых Windows. Минимальному набору кодов должны соответствовать команды, поступающие не только от клавиатуры, но и от всех других устройств ввода, в том числе от мыши. Таким образом, если какая-нибудь фирма выпустит четырехкнопочную мышь, то это не должно вас беспокоить, поскольку производитель сам позаботится о том, чтобы команды от дополнительных кнопок соответствовали стандартным кодам Windows.

2.2. Управление графическим выводом

Особенность создания приложений в среде Windows заключается в том, что здесь применяются специальные методики программирования и используется своя терминология. Последнюю можно разделить на две большие категории: терминология, связанная с пользовательским интерфейсом (меню, диалоговые окна, значки и т. д.), и терминология, относящаяся непосредственно к программированию (сообщения, вызовы функций и т. д.). Рассмотрим ниже эти термины и понятия.

2.2.1. Окно и его компоненты

Окно – это специальная прямоугольная область экрана. Все элементы окна, его размер и внешний вид контролируются открывающей его программой. Каждый щелчок пользователя на каком-нибудь элементе окна вызывает ответные действия приложения. Многозадачность в Windows заключается, в частности, в возможности одновременно открывать окна сразу нескольких приложений или нескольких окон одного приложения. Активизируя с помощью мыши или клавиатуры то или иное окно, пользователь дает системе по-

нять, что последующие команды и данные следует направлять именно этому окну.

Стандартный внешний вид окон всех приложений Windows и предсказуемость работы различных их компонентов позволяют пользователям чувствовать себя уверенно с новыми приложениями и легко разбираться в принципах их работы.

Рамка. Обычно каждое окно заключается в небольшую рамку. Функции рамки сводятся к отделению окна от остальных частей экрана. Как правило, рамка является и средством масштабирования. Размер окна приложения можно изменить, если поместить указатель мыши на рамку и перетащить его, удерживая нажатой левую кнопку мыши.

Строка заголовка. Имя приложения, которому принадлежит открытое окно, отображается в строке заголовка в верхней части окна. Строка заголовка является обязательным элементом всех окон приложений и позволяет пользователю легко определить, какому приложению принадлежит какое окно, если в системе запущено сразу несколько приложений. Строка заголовка активного окна выделяется альтернативным цветом, чтобы его легко можно было отличить от неактивных окон.

Значок приложения. Другим обязательным элементом любого окна является расположенный в его верхнем левом углу значок приложения. Этот значок обычно представляет собой маленький логотип приложения. Щелчок на значке приводит к открытию системного меню.

Системное меню. Системное меню открывается при щелчке мышью на значке приложения. В нем представлены стандартные команды управления окном: *Restore* (Восстановить), *Move* (Переместить), *Size* (Размер), *Minimize* (Свернуть), *Maximize* (Развернуть) и *Close* (Закрыть).

Кнопка свертывания. В правом верхнем углу большинства окон приложений имеется три кнопки. Крайняя левая кнопка предназначена для свертывания окна в кнопку на панели задач.

Кнопка развертывания/восстановления. Средняя кнопка в правом верхнем углу либо разворачивает окно на весь экран, либо восстанавливает прежние его размеры, если окно уже развернуто.

Кнопка закрытия. Крайняя правая кнопка в углу предназначена для закрытия приложения. После закрытия окна активным автоматически становится окно следующего приложения.

Вертикальная полоса прокрутки. В некоторых случаях окно приложения может содержать вертикальную полосу прокрутки, которая располагается по правому краю окна. В верхней и нижней частях полосы находятся кнопки со стрелками, направленными соответственно вверх и вниз. Вдоль самой полосы располагается бегунок. Положение бегунка показывает, какая часть окна или документа отображается в данный момент на экране. Перемещая бегунок, можно выбрать нужную часть многостраничного документа. Щелчок мышью на кнопке со стрелкой приведет к смещению содержимого

окна на одну строку вверх или вниз, а щелчок на свободном пространстве выше или ниже бегунка – на одну экранную страницу вверх или вниз.

Горизонтальная полоса прокрутки. Окно может также быть оснащено горизонтальной полосой прокрутки, которая размещается по нижнему краю окна и работает аналогично вертикальной полосе прокрутки. Горизонтальная полоса предназначена для выведения на экран частей документов, состоящих из большого числа столбцов. Щелчок мышью на кнопках со стрелками приводит к смещению содержимого окна на один столбец влево или вправо. Щелчок на областях между стрелками и бегунком смещает изображение на одну экранную страницу влево или вправо.

Строка меню. В окнах большинства приложений сразу под строкой заголовка находится строка меню, содержащая наборы команд и опций программы. Обычно для выбора команд меню используется мышь, хотя эти действия можно выполнить и с помощью клавиатуры. Каждому элементу меню, как правило, соответствует клавиша быстрого вызова, выделенная подчеркиванием в названии элемента. Чтобы выбрать данный элемент, нужно нажать соответствующую клавишу в сочетании с клавишей [Alt]. Так, комбинация клавиш [Alt+F] открывает меню File.

Рабочая область. Рабочая область обычно занимает большую часть окна. Именно в эту область программа выводит результаты своей работы.

2.2.2. Класс окна. Функция окна. Цикл обработки сообщений

Чтобы два окна одной и той же программы выглядели и работали совершенно одинаково, они оба должны базироваться на общем классе окна. В приложениях, написанных на C++, класс окна регистрируется программой в процессе инициализации. При создании окна класс указывается в качестве параметра функции CreateWindow(). Зарегистрированный новый класс становится доступным для всех программ, запущенных в данный момент в системе. В случае использования библиотеки MFC вся работа по регистрации классов окон выполняется автоматически, что существенно облегчает работу программиста.

Благодаря тому что окна приложения создаются на основе общего базового класса, значительно сокращается объем информации, которую при этом следует указывать. Поскольку класс окна содержит в себе описания элементов, общих для всех окон данного класса, нет необходимости повторять эти описания при создании каждого нового окна. К тому же все окна одного класса используют одну оконную процедуру, что также позволяет избежать дублирования кода.

Поскольку архитектура Windows-программ основана на принципе сообщений, все эти программы содержат некоторые общие компоненты. Обычно их приходится в явном виде включать в исходный код. При использовании библиотеки MFC это происходит автоматически; нет необходимости тратить время и усилия на их написание. Рассмотрим назначение этих компонентов.

Функция WinMain(). Все Windows-программы начинают выполнение с вызова функции WinMain(). При традиционном методе программирования это нужно делать явно. С использованием библиотеки MFC такая необходимость отпадает, но функция все-таки существует.

Функция окна. Все Windows-программы должны содержать специальную функцию, которая не используется в самой программе, но вызывается самой операционной системой. Эту функцию обычно называют функцией окна или процедурой окна. Она вызывается Windows, когда системе необходимо передать сообщение в программу. Именно через нее осуществляется взаимодействие между программой и системой. Функция окна передает сообщение в своих аргументах. Помимо принятия сообщения от Windows функция окна должна вызывать выполнение действия, указанного в сообщении. Конечно, программа не обязана отвечать на все сообщения, посылаемые Windows. Поскольку их могут быть сотни, то большинство сообщений обычно обрабатывается самой системой, а программе достаточно поручить Windows выполнить действия, предусмотренные по умолчанию. При использовании библиотеки MFC такая функция создается автоматически. Но в любом случае, если сообщение получено, то программа должна выполнить некоторое действие.

Именно способ взаимодействия с операционной системой через сообщения диктует общий принцип построения всех программ для Windows, написанных как с использованием MFC, так и без нее.

Цикл сообщений. Итак, Windows взаимодействует с программой, посылая ей сообщения. Все приложения Windows должны организовать так называемый цикл сообщений (обычно внутри функции WinMain()). В этом цикле каждое необработанное сообщение должно быть извлечено из очереди сообщений данного приложения и передано назад в Windows, которая затем вызывает функцию окна программы с данным сообщением в качестве аргумента.

В традиционных Windows-программах необходимо самостоятельно создавать и активизировать такой цикл. При использовании MFC это также выполняется автоматически. Однако важно помнить, что цикл сообщений все же существует. Он является неотъемлемой частью любого приложения Windows.

Процесс получения и обработки сообщений является сложным, но тем не менее ему должны следовать все Windows-программы. При использовании библиотеки MFC большинство частных деталей скрыто от программиста, хотя они и продолжают неявно присутствовать в программе.

Класс окна. Каждое окно в Windows-приложении характеризуется определенными атрибутами, называемыми классом окна. (Здесь понятие «класс» не идентично используемому в C++. Оно, скорее, означает стиль или тип.)

В традиционной программе класс окна должен быть определен и зарегистрирован прежде, чем будет создано окно. При регистрации необходимо сообщить Windows, какой вид должно иметь окно и какую функцию оно выполняет. В то же время регистрация класса окна еще не означает создание самого окна. Для этого требуется выполнить дополнительные действия. При использовании библиотеки MFC создавать собственный класс окна нет необ-

ходимости. Вместо этого можно работать с одним из заранее определенных классов, описанных в библиотеке.

2.2.3. Графические объекты, используемые в окнах

Примерами графических объектов, с которыми можно обращаться как с единым целым и которые выступают элементами пользовательского интерфейса, являются строка меню, полосы прокрутки, кнопки и т. д. Рассмотрим наиболее широко используемые графические объекты Windows.

Значки. Значками называются маленькие графические изображения, выполняющие опознавательную функцию. Так, значки приложений на панели задач позволяют легко определить, какие программы в настоящий момент запущены, даже если названия программ не отображаются целиком. Значки могут быть очень полезны в самих приложениях, поскольку с их помощью можно привлекать внимание пользователей к сообщениям об ошибках и различным предупреждениям. В Windows входит набор стандартных значков, в частности стилизованные знак вопроса и восклицательный знак, а также ряд других. С помощью встроенного в компилятор Visual C++ редактора ресурсов вы можете создавать собственные значки.

Указатели мыши. Указатели мыши также являются графическими объектами, используемыми для отслеживания перемещения мыши. Вид указателя может меняться в зависимости от выполняемого задания и состояния системы. Например, стандартный указатель «стрелка» меняет свой вид на изображение песочных часов в том случае, если система занята. С помощью встроенного в компилятор Visual C++ редактора ресурсов вы можете создавать собственные указатели мыши.

Текстовые курсоры. Курсоры предназначены для указания места, куда следует осуществлять ввод текстовых данных. Отличительной особенностью курсоров является их мерцание. В большинстве текстовых редакторов и в полях диалоговых окон в качестве курсора применяется I-образный указатель мыши. Причем в Windows нет (в отличие от значков и указателей) коллекции готовых курсоров.

Окна сообщений. Окна сообщений представляют собой разновидность диалоговых окон, содержащих строку заголовка, значок и текст сообщения.

В приложении при вызове специальной функции указывается текст заголовка окна, текст сообщения, какой из стандартных значков Windows использовать (если это необходимо) и какой набор кнопок выводить. В частности, можно вызывать окна с такими комбинациями кнопок: Abort/Retry/Ignore, OK, Yes/No, Yes/No/Cancel, OK/Cancel и Retry/Cancel. Обычно в окнах сообщений отображаются такие стандартные значки, как IconHand, IconQuestion, IconExclamation, IconAsterisk и некоторые другие.

Диалоговые окна. Диалоговые окна содержат наборы различных элементов управления и предназначены для предоставления пользователю возможности устанавливать опции и параметры программы, которой принадле-

жит данное окно. Внешний вид диалогового окна разрабатывается с помощью редактора ресурсов компилятора Visual C++.

Шрифты. Шрифт в Windows – это графический ресурс, содержащий набор символов определенного типа. Существует набор функций, с помощью которых можно манипулировать начертанием символов для получения форматированного текста. В приложениях можно использовать различные стандартные шрифты, такие как Courier, TimesNewRoman и другие, а также пользовательские шрифты. Встроенные функции позволяют на базе основного шрифта получать полужирное начертание, курсив, подчеркнутый текст, изменять размер шрифта. Причем внешний вид шрифта не зависит от типа устройства, на которое выводится текст.

Точечные рисунки. Точечные рисунки (bitmap) представляют собой точную копию части экрана, снятую попиксельно. Тот факт, что изображение является точным образом экрана, устраняет необходимость в каких бы то ни было дополнительных преобразованиях, что существенно сокращает время вывода изображения на экран. В Windows точечные рисунки наиболее широко применяются для двух целей. Во-первых, они служат изображениями всевозможных кнопок и значков, например стрелок полос прокрутки и кнопок панелей инструментов. Другой областью применения точечных рисунков являются кисти, с помощью которых рисуются и заполняются цветом различные геометрические фигуры на экране. Точечные рисунки можно создавать и модифицировать с помощью встроенного редактора ресурсов.

Перья. Перья предназначены для рисования геометрических фигур и различных контуров и характеризуются тремя основными параметрами: ширина линии, стиль (точечный, штрихпунктирный, непрерывный) и цвет. Существует два готовых пера: одно для рисования черных линий, другое для рисования белых. С помощью специальных функций вы можете создавать собственные перья.

Кисти. Кисти предназначены для заливки объектов цветом, выбранным из заданной палитры. Минимальный размер кисти – 8x8 пикселей. Кисть также характеризуется тремя *параметрами:* размером, шаблоном заливки и цветом. Заливка может быть сплошной, штриховой, диагональной или представлять собой узор, заданный пользователем.

3. Основные этапы разработки Windows-приложений

3.1. Мастер приложений AppWizard и библиотека MFC

В связи с тем, что сегодня уровень сложности программного обеспечения очень высок, разработка приложений Windows с использованием только какого-либо языка программирования (например языков C или C++) значительно затрудняется.

Чтобы облегчить работу программиста практически все современные компиляторы с языка C++ содержат специальные библиотеки классов [6]. Такие библиотеки включают в себя практически весь программный интерфейс Windows и позволяют пользоваться при программировании средствами более высокого уровня, чем обычные вызовы функций.

Современные интегрированные средства разработки приложений Windows позволяют автоматизировать процесс создания приложения. Для этого используются генераторы приложений. Программист отвечает на вопросы генератора приложений и определяет свойства приложения – поддерживает ли оно многооконный режим, технологию OLE, трехмерные органы управления, справочную систему. Генератор приложений создаст приложение, отвечающее требованиям, и предоставит исходные тексты. Пользуясь им как шаблоном, программист сможет быстро разрабатывать свои приложения.

Подобные средства автоматизированного создания приложений включены и в компилятор Microsoft Visual C++. Они предполагают создание приложений различных типов. Для создания Windows-приложения, использующего библиотеку классов MFC, применяется средство MFC AppWizard (exe). Заполнив несколько диалоговых панелей, можно указать характеристики приложения и получить его тексты, снабженные обширными комментариями.

MFC – это базовый набор (библиотека) классов, написанных на языке C++ и предназначенных для упрощения и ускорения процесса программирования для Windows. Прикладную часть приложения программист разрабатывает самостоятельно. Исходный текст приложения, созданный MFC AppWizard, станет только основой, к которой нужно подключить остальное.

3.1.1. Интерфейс вызовов функций в Windows

Доступ приложений к системным ресурсам осуществляется через целый ряд системных функций, называемых прикладным программным интерфейсом или API (Application Programming Interface).

Для взаимодействия с Windows приложение запрашивает функции API, с помощью которых реализуются все необходимые системные действия, такие как выделение памяти, вывод на экран, создание окон и т. п.

Библиотека MFC инкапсулирует многие функции API. Хотя программам и разрешено обращаться к ним напрямую, все же чаще это будет выполняться через соответствующие функции-члены. Как правило, функции-члены

либо аналогичны функциям API, либо непосредственно обращаются к нужной части интерфейса. Функции API содержатся в библиотеках динамической загрузки (DLL), которые загружаются в память только в тот момент, когда к ним происходит обращение, т. е. при выполнении программы.

Динамическая загрузка обеспечивает ряд существенных преимуществ.

Во-первых, поскольку практически все программы используют API-функции, то благодаря DLL-библиотекам существенно экономится дисковое пространство, которое в противном случае занималось бы большим количеством повторяющегося кода, содержащегося в каждом из исполняемых файлов.

Во-вторых, изменения и улучшения в Windows-приложениях сводятся к обновлению только содержимого DLL-библиотек. Уже существующие тексты программ не требуют перекомпиляции.

3.1.2. Взаимодействие программ и Windows

В Windows система вызывает программу. Это осуществляется следующим образом: программа ожидает получения сообщения от Windows. Когда это происходит, то выполняется некоторое действие. После его завершения программа ожидает следующего сообщения.

Windows может посылать программе сообщения множества различных типов. Например, каждый раз при щелчке мышью в окне активной программы посылается соответствующее сообщение. Другой тип сообщений посылается, когда необходимо обновить содержимое активного окна, сообщения посылаются также при нажатии клавиши, если программа ожидает ввода с клавиатуры. По отношению к программе сообщения появляются случайным образом. Windows-программы похожи на программы обработки прерываний: невозможно предсказать, какое сообщение появится в следующий момент.

Структура Windows-программ отличается от структуры программ других типов. Это вызвано двумя обстоятельствами: во-первых, способом взаимодействия между программой и Windows; во-вторых, правилами, которым следует подчиняться для создания стандартного интерфейса Windows-приложения. Значительная часть кода Windows-приложения предназначена именно для организации интерфейса с пользователем.

Чтобы отойти от философии создания традиционного Windows-интерфейса, должны быть достаточно веские основания. Если программист собирается писать приложения для Windows, то он должен дать пользователям возможность работать с привычным интерфейсом и руководствоваться стандартной методикой разработки.

3.2. Основные этапы построения каркаса приложения

Мастер создания приложений AppWizard позволяет создавать различные типы приложений, но обычно большинством программистов используются исполняемые программы (файл приложения с расширением .exe). Кро-

ме того, желательно получить от AppWizard готовые фрагменты программного кода – классы, объекты, функции, которые присутствуют едва ли не в каждой стандартной программе. Для создания программы подобного типа необходимо выбрать File\New, а затем – вкладку Projects в окне New, как это показано на рис. 3.1.

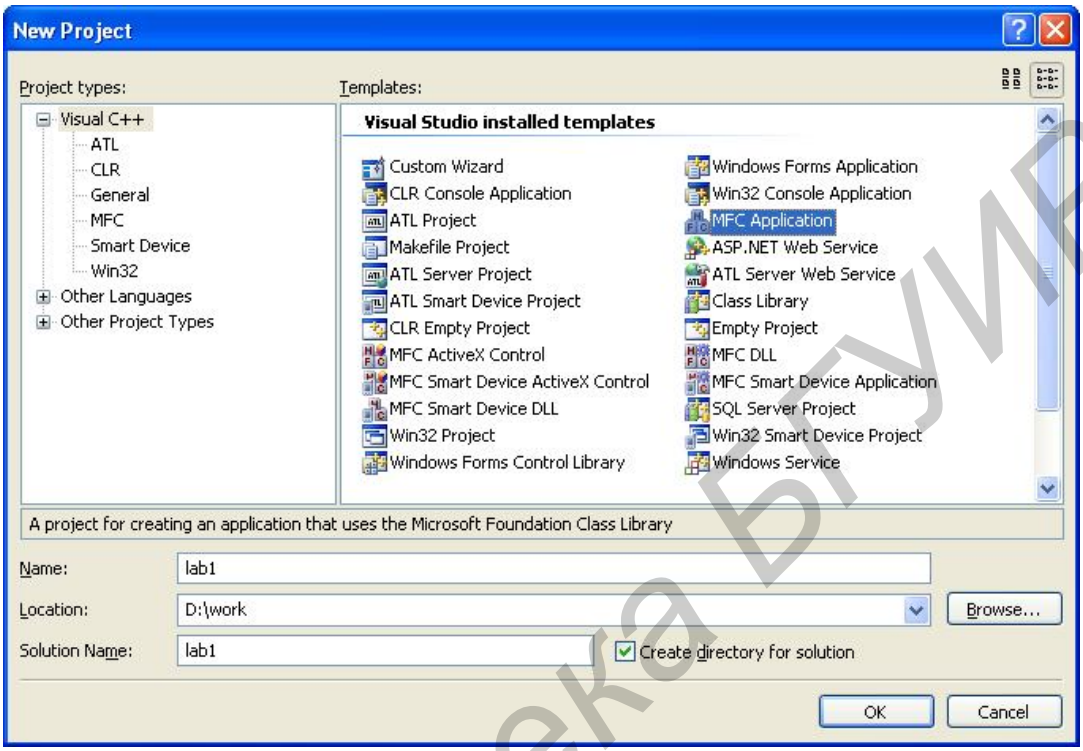

Рис. 3.1. Выбор типа приложения

В левой части окна находится список возможных типов проектов. Для создания типового приложения необходимо выбрать MFC Application. Также необходимо указать имя проекта в поле Name, а в поле Location – каталог, в котором будет находиться проект.

Данный тип проекта использует библиотеку классов Microsoft Foundation Classes (MFC). На каждом этапе программист может изменить некоторые параметры создаваемого приложения. Рассмотрим более подробно этапы создания приложения AppWizard. На следующем этапе следует настраивать опции приложения, представленные в списке Overview (рис. 3.2).

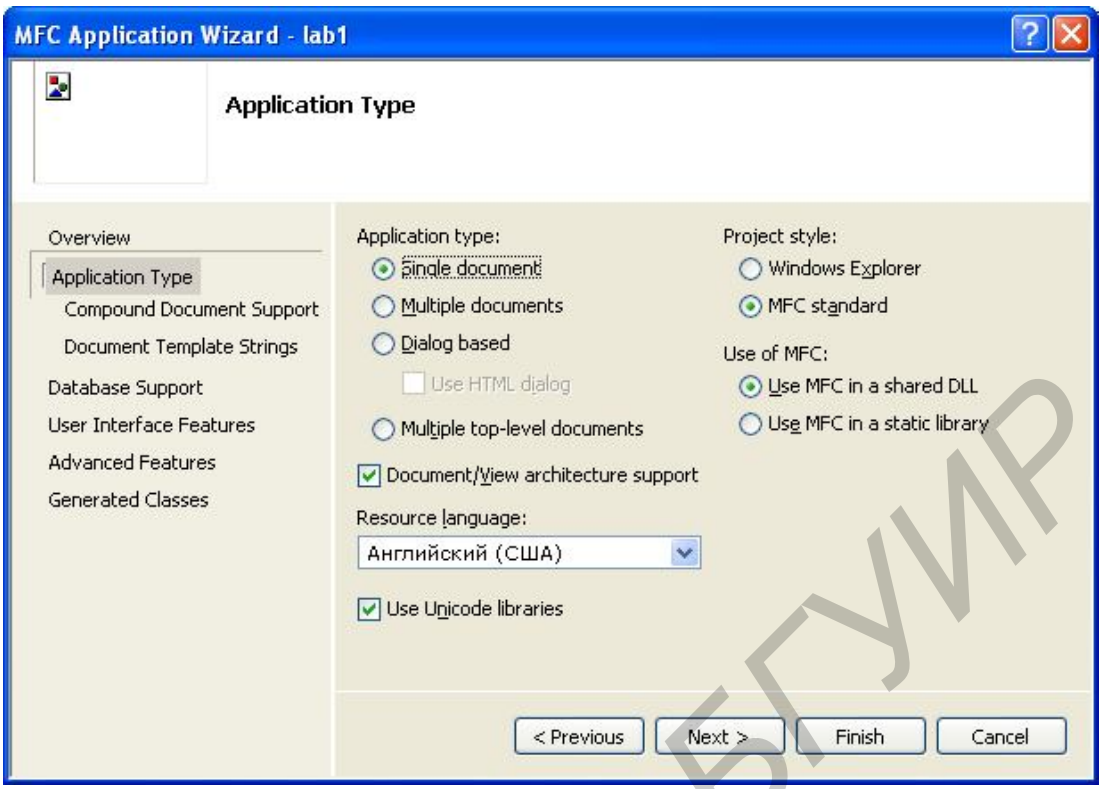

Рис. 3.2. Выбор варианта интерфейса пользователя

3.2.1. Выбор типа приложения

Application type. Первое, что должен определить программист, приступая к работе в AppWizard, – сколько документов будет поддерживать будущее приложение, т. е. будет ли оно многодокументным приложением (Multiply documents), однодокументным приложением (Single document) или простым диалоговым приложением (Dialog based). Для каждого из этих типов приложений AppWizard создает различные классы:

– SDI-приложение (SDI – Single Document Interface, интерфейс с единственным документом) позволяет в каждый момент времени иметь открытым только один документ, однако количество открытых окон не ограничено. Примером может служить известный каждому редактор Notepad;

– MDI-приложение (MDI – Multiple Document Interface, дословно – «многодокументный интерфейс») может одновременно держать открытыми несколько документов, каждый из которых представлен отдельным файлом, примеры – Excel, Word и другие аналогичные приложения. Такие приложения обязательно имеют в главном меню пункт Window, а в меню File – пункт Close;

– простое диалоговое приложение, как правило, вообще не открывает документов. Такие приложения не имеют меню.

Ниже этой группы переключателей в диалоговом окне находится флажок *Document/View architecture support* (поддержка архитектуры доку-

мент/представление). Пользователи, которые имеют большой опыт разработки приложений в среде Visual C++, могут отключить поддержку этой архитектуры мастером, но для большинства разработчиков она будет отнюдь не лишней. Поэтому в дальнейшем, если не будет оговорено особо, будем считать, что флажок Document/View architecture support установлен.

Еще ниже в диалоговом окне находится раскрывающийся список для выбора языка, который вы собираетесь использовать при написании текста программы. Если системный язык операционной среды, не заданный по умолчанию, English (United States), сделайте такой же выбор и в списке.

Опция Compound Document Support. Окно MFC AppWizard при этом будет выглядеть так, как показано на рис. 3.3.

На выбор предлагается пять вариантов поддержки:

– если не планируется создание OLE-приложения, выберите переключатель None (Никакой);

– если вы хотите, чтобы в приложении использовались связанные или внедренные объекты OLE (например такие, как документы Word или рабочие листы Excel), выберите переключатель Container (Контейнер);

– если планируется создание приложения, документы которого могли бы быть внедрены в другое приложение, но при этом само приложение не будет использоваться автономно, выберите переключатель Mini-server (Мини-сервер);

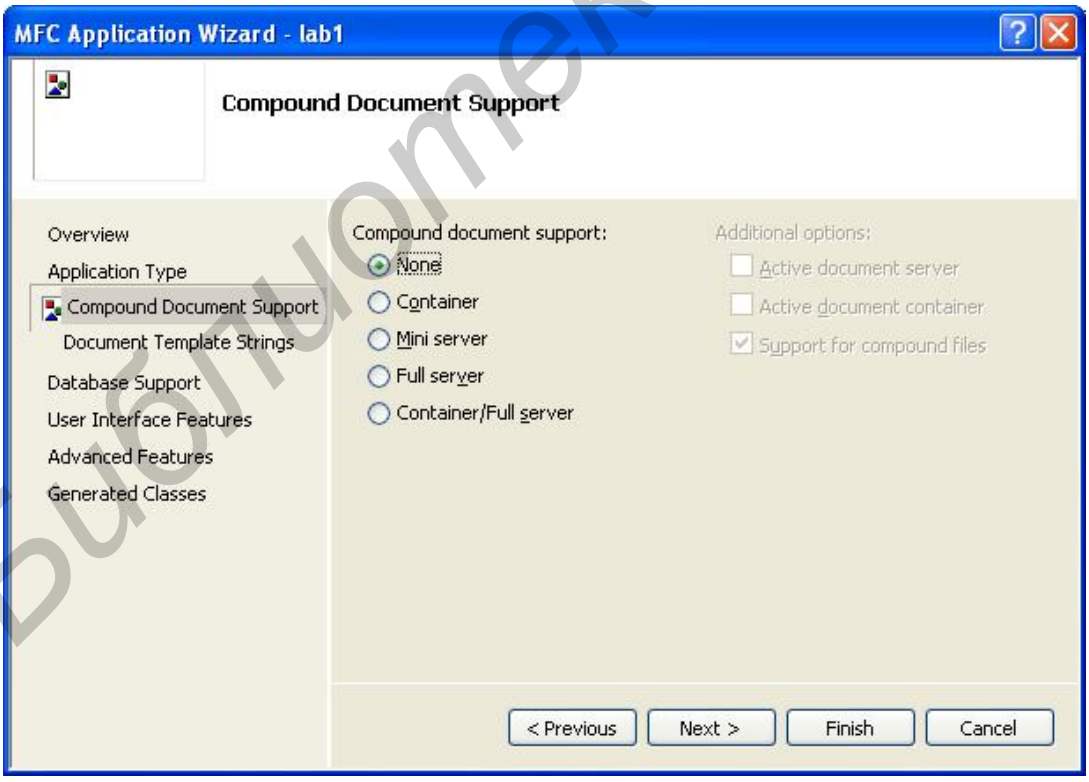

Рис. 3.3. Выбор варианта поддержки составных документов

– если ваше будущее приложение будет не только служить сервером для других приложений, но и сможет работать автономно, выберите переключатель Full-server (Полный сервер);

– если создаваемое приложение должно обладать способностью включать документы других приложений и само обслуживать их своими объектами, выбирайте переключатель Both container and server (и контейнер, и сервер).

Если вы выбрали какой-либо из вариантов поддержки составных документов, то придется поддерживать и составные файлы (compound files). Составные файлы содержат один или более объектов ActiveX и сохраняются на диске в особом формате, так что один из объектов может быть заменен без переписи всего файла. Таким образом удается сберечь довольно много времени.

3.2.2. Выбор поддержки баз данных

Опция Database Support. Окно MFC AppWizard при этом будет выглядеть так, как показано на рис. 3.4.

Здесь вам на выбор предлагаются четыре варианта уровня поддержки:

– если работа с базами данных в приложении не планируется, выберите переключатель None (Никакой);

– если предполагается иметь доступ к базам данных, но для этого не будут использованы классы просмотра, производные от CFormView, или нет необходимости в меню Record (Запись), выбирайте переключатель Header files only (Только файлы заголовков);

– если вы планируете разрабатывать классы просмотра базы данных в приложении как производные от CFormView и иметь меню Record, но не нуждаетесь в средствах сохранения-восстановления (сериализации) документов, выбирайте переключатель Database view without file support (Просмотр базы данных без поддержки операций с файлами). Записи в базе данных можно будет обновлять с помощью CRecordset – класса MFC доступа к базам данных;

– если помимо всего, что задано в предыдущем случае, вы планируете и сохранение-восстановление документов на диске (возможно, это будет одна из опций пользователя), выберите Database view with file support (Просмотр базы данных и поддержка операций с файлами).

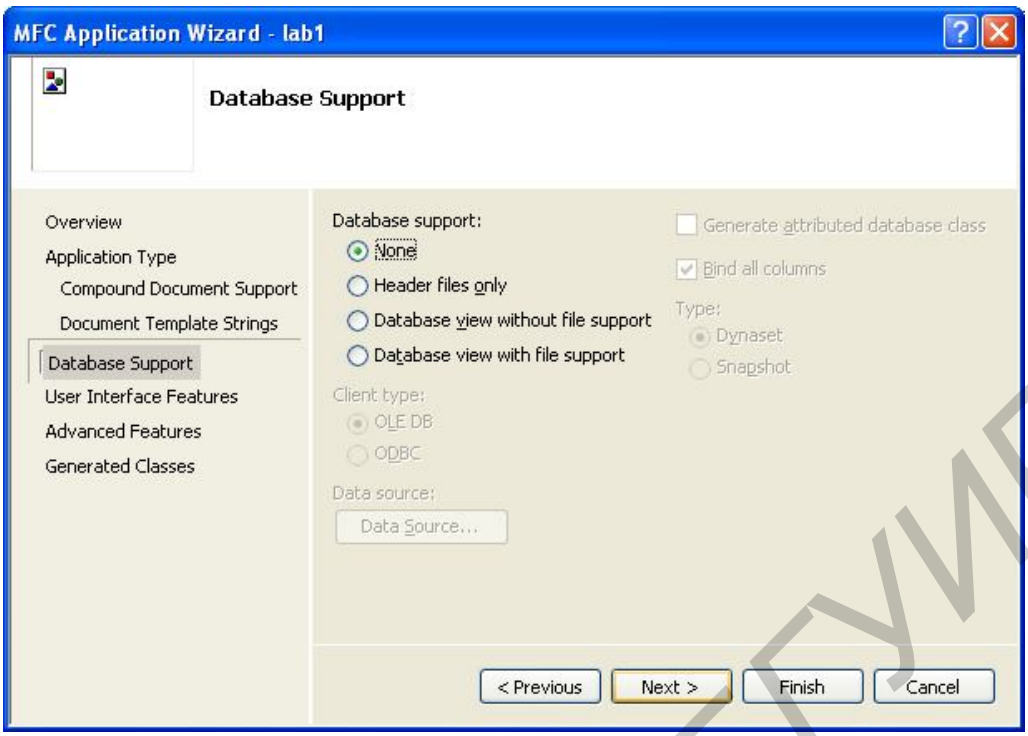

Рис. 3.4. Выбор варианта поддержки операции с базами данных

Если вы выбрали один из вариантов с использованием базы данных, в этом же окне задайте базу, которая будет источником данных. Для этого нужно щелкнуть на кнопке Data Source (Источник данных).

3.2.3. Установка опций пользовательского интерфейса

Опция User Interface Features. Окно MFC AppWizard при этом будет выглядеть так, как показано на рис. 3.5.

Это диалоговое окно содержит много переключателей-флажков, соответствующих предлагаемым опциям оформления:

– Toolbars (Вид панели инструментов). В приложение будет добавлена панель инструментов, которая может быть стиля «Standart docking» или «Browser style»;

– Initial Status bar (Панель состояния). В приложение будет добавлена панель состояния, в которой можно будет выводить подсказки соответственно выбранным пунктам меню и другие сообщения;

– Split Window (Использование разделения окна). При его установке в приложение включается весь программный код, необходимый для организации разделения окна приложения таким же образом, как это сделано, например, в редакторе программного кода из комплекта средств Visual Studio. Остальные элементы диалогового окна устанавливают параметры, определяющие оформление фрейма (рамки) главного окна приложения, а для MDI-приложений – фреймов дочерних окон (child frames). Фрейм является весьма важным элементом окна. Системное меню, строка заголовка, кнопки мини-

мизации и максимизации, собственно границы – все это свойства фрейма как объекта. Фрейм главного окна содержит всё SDI-приложение. MDI-приложение имеет несколько дочерних окон (по одному на каждый документ), которые размещаются в пределах главного окна;

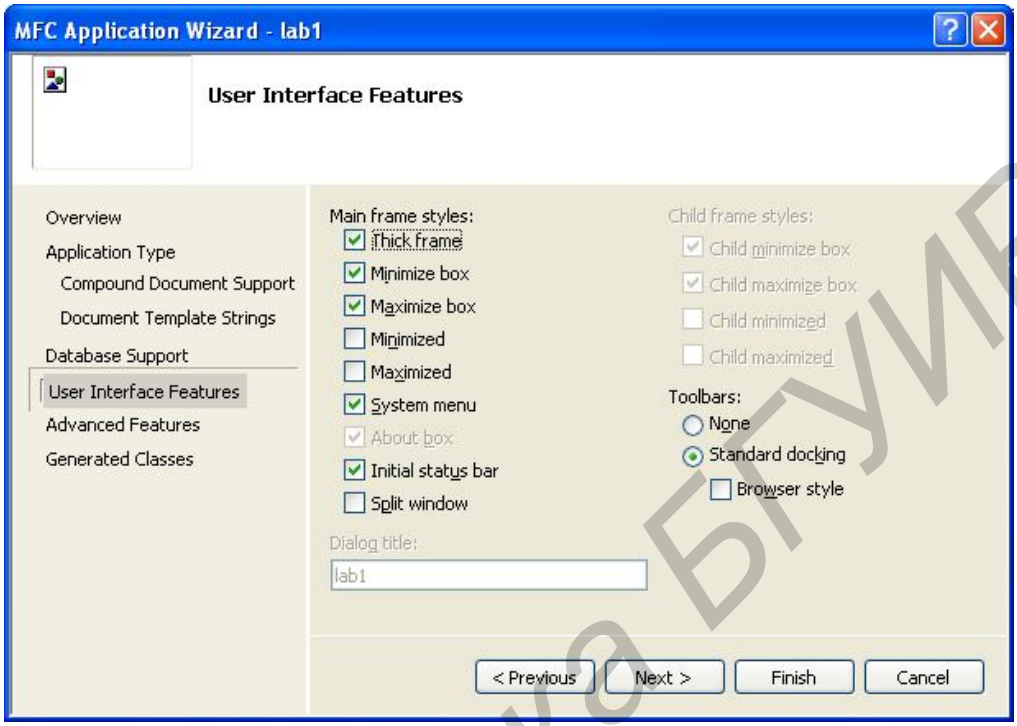

Рис. 3.5. Установка опций пользовательского интерфейса

– Thick frame (утолщенная рамка) – кромки окна утолщены, и можно будет изменять размеры окна стандартным для Windows способом;

– Minimize box (кнопка минимизации) – окно имеет кнопку минимизации в правой части строки заголовка;

– Maximize box (кнопка максимизации) – окно имеет кнопку максимизации в правой части строки заголовка;

– System menu (системное меню) – в левом верхнем углу окна будет установлена пиктограмма вызова системного меню;

– Minimized – при запуске приложения окно сворачивается в пиктограмму. Для SDI-приложений выбор этой опции не будет иметь никаких последствий;

– Maximized – при запуске приложения окно разворачивается на весь экран. Для SDI-приложений выбор этой опции не будет иметь никаких последствий.

Опция Advanced Features. Окно MFC AppWizard при этом будет выглядеть так, как показано на рис. 3.6:

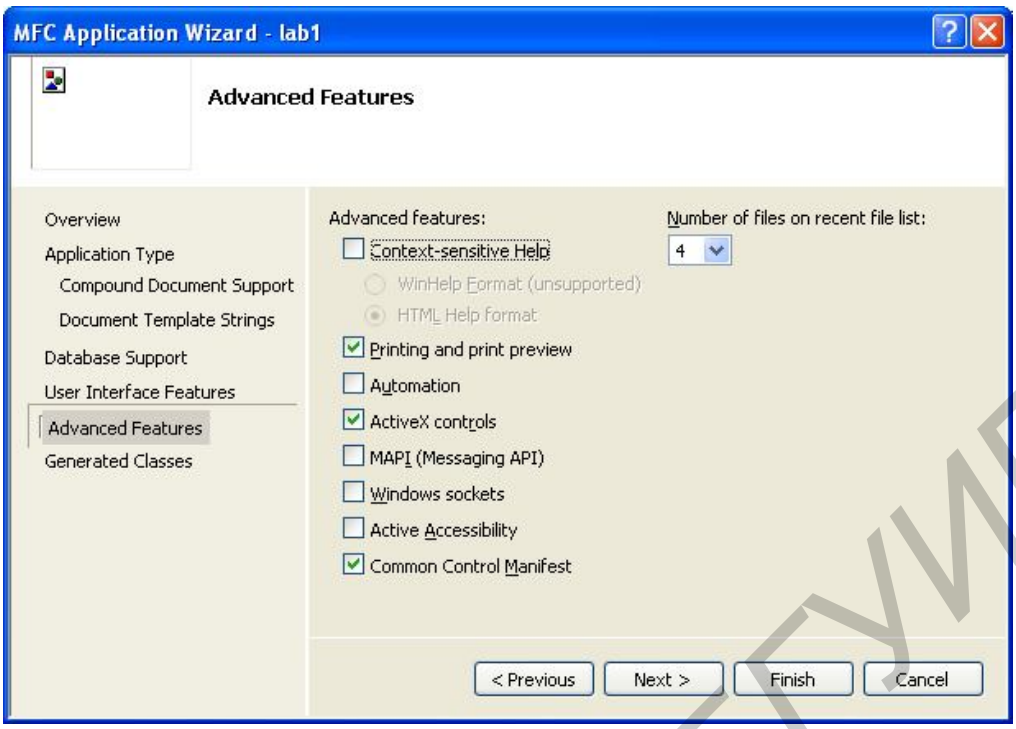

Рис. 3.6. Установка некоторых дополнительных опций ApplicationWizard

– Printing and print preview (Печать и предварительный просмотр печати). Приложение при выборе этой опции будет иметь пункты Print и Print preview в меню File, и AppWizard включит в приложение большую часть программного кода, связанного с выполнением этих операций;

– Context sensitive Help (Контекстная справка). Меню Help в приложении будет иметь опции Index и Using Help, а значительная часть программного кода, необходимого для организации контекстной справки в приложении, будет включена в него мастером AppWizard;

– MAPI (Messaging API – почтовый интерфейс). При установке этой опции приложение сможет обмениваться сообщениями по электронной почте;

– Windows Sockets. Если эта опция будет установлена, приложение сможет иметь непосредственный доступ к Internet через такие протоколы, как FTP и HTTP (протокол World Wide Web). Можно создать Internet-программу и без поддержки Windows Sockets, если использовать классы WinInet;

– Number of files on recent file list. Можно установить длину списка последних открываемых файлов в поле меню File создаваемого приложения. По умолчанию этот параметр имеет значение 4, и менять его не рекомендуется без очень весомых причин.

Подтверждение имен классов и файлов. Окно MFC AppWizard при этом будет выглядеть так, как показано на рис. 3.7. AppWizard использует имя проекта (в данном случае – lab1) для формирования имен классов и имен файлов. Нет никакой нужды их изменять. Если в приложении используются классы представления, можно изменить имя класса, наследниками которого

являются вновь создаваемые классы. По умолчанию базовым является CView, но многие разработчики предпочитают CScrollView или CEditView. После завершения работы в диалоговом окне необходимо нажать на кнопку Finish.

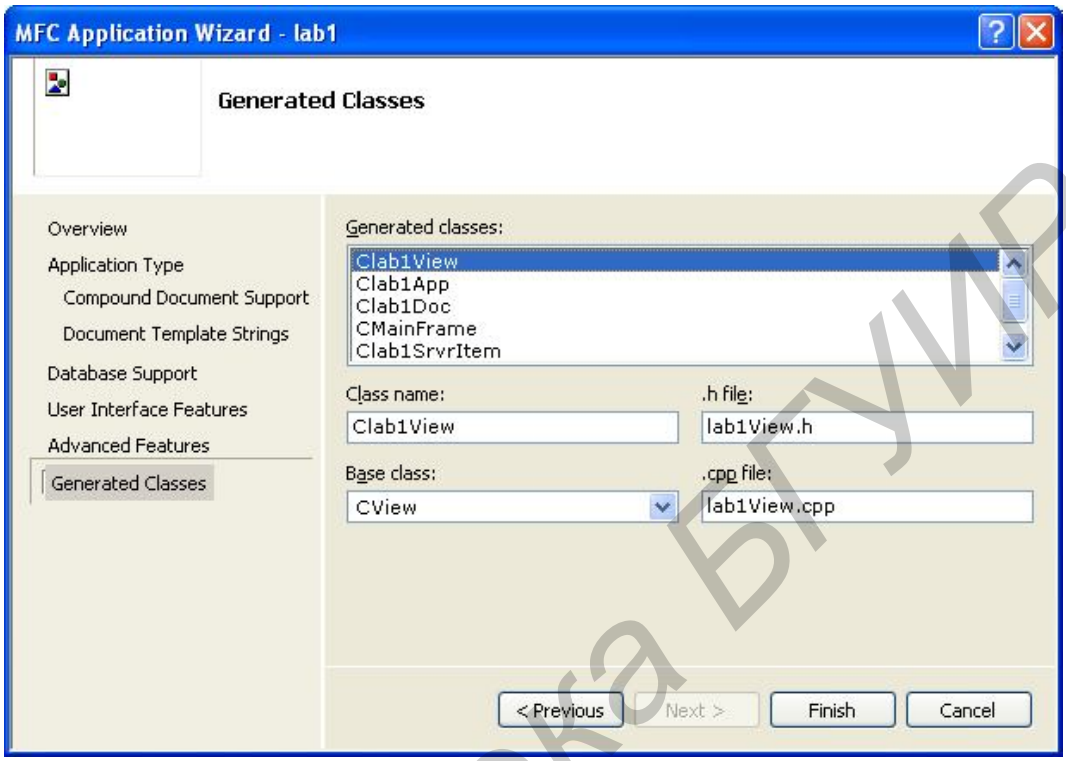

Рис. 3.7. Подтверждение имен классов и файлов на последнем этапе создания типового приложения

После того как вы щелкнете на Finish, AppWizard покажет вам в окне New Project Information информацию о новом проекте. Если что-либо не устраивает, можно вернуться, нажав на кнопку Cancel, и затем последовательно двигаться в обратном порядке по окнам этапов настройки, пока не будет найдено то окно, в котором возможно изменение данной настройки. После уточнения настройки можно повторить путь по шагам AppWizard'а либо сразу принять оставшиеся установки. После чего можно опять взглянуть на окно New project information и дать согласие на генерацию классов. AppWizard создаст необходимые классы и ресурсы.

4. Архитектура «документ/представление»

4.1. Классы документа и представления

4.1.1. Класс документа

Формируя текст программы приложения с помощью AppWizard, разработчик имеет возможность оснастить свое детище всеми модными аксессуарами коммерческого продукта для Windows – панелью инструментов, строкой состояния, контекстным окном указателя, разнообразными меню и даже окном сообщения об авторских правах. Для того чтобы создать приложение, которое не только хорошо выглядит на экране, но и делает что-то полезное, придется вмешаться в текст программы, которую подготовил AppWizard.

Наиболее существенным изменениям подвергнется часть программы, связанная с документом – информацией, которую пользователь может сохранять в процессе работы с приложением и затем считывать, и с представлением – средствами представления этой информации пользователю в процессе выполнения приложения. Положенная в основу MFC концепция документ/представление позволяет отделить данные от средств, с помощью которых пользователь имеет возможность просмотреть эти данные и манипулировать ими. Объекты-документы ответственны за хранение, загрузку и выгрузку данных, а объекты-представления, которые подчас представляют собой те же окна, позволяют пользователю просматривать данные на экране и редактировать их соответственно логике работы приложения. Рассмотрим основы работы MFC в части реализации концепции документ/представление [7].

Создавая SDI- и MDI-приложения, AppWizard изначально закладывает в них средства, ориентированные на реализацию концепции документ/представление. Это означает, что AppWizard формирует класс, производный от CDocument, и передает ему определенные задачи. Он также формирует класс представления, производный от CView, которому передает другие задачи. Чтобы создать с помощью AppWizard простейшее приложение, выполните следующие действия.

Выберите File→New, затем вкладку Projects. Установите имя проекта App1 и соответствующие каталоги для файлов проекта. Выберите опцию Multiple documents, остальные настройки – по умолчанию.

После щелчка на кнопке OK ClassWizard сформирует проект. В окне ClassView просмотрите список классов приложения. Создано шесть классов: CAboutDlg, CApp1App, CApp1Doc, CApp1View, CChiIdFrame и CMainFrame.

Класс CApp1Doc представляет документ и содержит структуру данных, которыми может оперировать приложение. Организовать хранение данных в классе можно включением в него соответствующих членов-переменных. Текст файла заголовка, который AppWizard сформировал для класса CApp1Doc, представлен в листинге 4.1.

Листинг 4.1. Интерфейс класса CApplDoc.

```
// App1Doc.h :
////////////////////////////////////////////////////////////
#if !defined
(AFX_APP1DOC_H__528CF70C_0F96_11D7_BBD4_00C0F6B2220A__INCLUDED_)
#define
AFX_APP1DOC_H__528CF70C_0F96_11D7_BBD4_00C0F6B2220A__INCLUDED_
#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000
class CApp1Doc : public CDocument
{
protected:
     CApp1Doc();
     DECLARE_DYNCREATE(CApp1Doc)
// Attributes
public:
// Операции.
public:
     //{{AFX_VIRTUAL(CApp1Doc)
     public:
     virtual BOOL OnNewDocument();
     virtual void Serialize(CArchive& ar);
     //}}AFX_VIRTUAL
// Реализация
public:
     virtual ~CApp1Doc();
#ifdef _DEBUG
     virtual void AssertValid() const;
     virtual void Dump(CDumpContext& dc) const;
#endif
protected:
// Карта сообщений
protected:
     //{{AFX_MSG(CApp1Doc)
// ВНИМАНИЕ! ! Здесь ClassWizard будет добавлять и
// удалять функции-члены. НЕ РЕДАКТИРУЙТЕ текст в этих блоках!
     //}}AFX_MSG
     DECLARE_MESSAGE_MAP()
};
}
```

Почти в самом начале есть секция Атрибуты, за которой следует ключевое слово public. Именно здесь вам и нужно будет вставить объявления членов-переменных, в которых планируется хранить данные приложения. В приложении, которое будет рассмотрено дальше нужно будет сохранять массив объектов класса CPoint. Этот массив объявляется как член класса документа:

```
// Attributes
public:
CPoint m_points [100];
```

CPoint – это класс MFC, который инкапсулирует информацию, имеющую отношение к точке на экране, в частности координаты этой точки x и y.

В тексте файла заголовка обратите внимание также на то, что класс CApp1Doc имеет две виртуальные функции-члены – OnNewDocument() и Serialize(). MFC вызывает функцию OnNewDocument(), как только пользователь выберет команду File=>New (или соответствующую пиктограмму на панели инструментов, если таковая присутствует в приложении). Эту функцию можно использовать для выполнения всех инициализаций, необходимых новой порции данных. В SDI-приложении в таком случае закрывается открытый документ, и новый пустой документ загружается в тот же самый объект класса. В MDI-приложении открывается новый пустой документ (создается новый экземпляр класса документа) в дополнение к уже существующему. Функция Serialize() используется для загрузки в файл и выгрузки из него данных, хранящихся в членах-переменных объекта класса документа.

В листинге 4.2 представлен файл App1Doc.cpp.

Листинг 4.2. Файл App1Doc.cpp.

```
// App1Doc.cpp : implementation of the CApp1Doc class
#include "stdafx.h"
#include "App1.h"
#include "App1Doc.h"
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
/////////////////////////////////////////////////////////////
// CApp1Doc
IMPLEMENT_DYNCREATE(CApp1Doc, CDocument)
BEGIN_MESSAGE_MAP(CApp1Doc, CDocument)

END_MESSAGE_MAP()
/////////////////////////////////////////////////////////////
// CApp1Doc construction/destruction
CApp1Doc::CApp1Doc()
{
// TODO: add one-time construction code here
}
CApp1Doc::~CApp1Doc()
{
}
BOOL CApp1Doc::OnNewDocument()
{    if (!CDocument::OnNewDocument())
          return FALSE;
     // TODO: add reinitialization code here
     // (SDI documents will reuse this document)
     return TRUE;
```

```
}
/////////////////////////////////////////////////////////////////
// CApp1Doc serialization
void CApp1Doc::Serialize(CArchive& ar)
{    if (ar.IsStoring())
     {
          // TODO: add storing code here
     }
     else
     {
          // TODO: add loading code here
     }
}
/////////////////////////////////////////////////////////////////
// CApp1Doc diagnostics
#ifdef _DEBUG
void CApp1Doc::AssertValid() const
{
     CDocument::AssertValid();
}
void CApp1Doc::Dump(CDumpContext& dc) const
{
     CDocument::Dump(dc);
}
#endif //_DEBUG
/////////////////////////////////////////////////////////////////
// CApp1Doc commands
```

В файле App1Doc.cpp представлены каркасы основных методов класса документа, которые следует формировать для корректной работы с данными документа.

4.1.2. Класс представления

Класс представления отвечает за вывод на экран данных, хранящихся в объекте класса документа, и позволяет пользователю модифицировать эти данные. Объект класса представления содержит указатель на объект класса документа, который используется для доступа к членам-переменным этого класса, где собственно и хранятся данные.

Большинство программистов, имеющих дело с MFC, включают в класс документа открытые (public) члены, с тем чтобы не затруднять доступ к ним из объекта класса представления. Классический объектно-ориентированный подход, однако, требует включать в класс закрытые (private) или защищенные (protected) члены-переменные и открытые члены-функции считывания и модификации этих переменных.

Рассмотрим текст файла заголовка, который AppWizard сформировал для класса CApplView (листинг 4.3).

Листинг 4.3. Файл App1View.h.

```
// App1View.h : interface of the CApp1View class
…
class CApp1View : public CView
{
protected: // create from serialization only
     CApp1View();
     DECLARE_DYNCREATE(CApp1View)
// Attributes
public:
     CApp1Doc* GetDocument();
// Operations
public:
     //{{AFX_VIRTUAL(CApp1View)
     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);
     //}}AFX_VIRTUAL
// Implementation
public:
     virtual ~CApp1View();
#ifdef _DEBUG
     virtual void AssertValid() const;
     virtual void Dump(CDumpContext& dc) const;
#endif
protected:
// Сформированные функции карты сообщений
protected:
     //{{AFX_MSG(CApp1View)
     //}}AFX_MSG
     DECLARE_MESSAGE_MAP()
};
```

Почти в самом начале текста имеется секция открытых атрибутов, в которой объявлена функция GetDocument(), возвращающая указатель на объект класса CApplDoc. Если, работая с классом представления, вы пожелаете получить указатель на объект класса документа, нужно будет вызвать эту функцию, и она вернет вам требуемый указатель.

В распространяемой (release) версии приложения функция GetDocument() объявлена как встроенная (inline). В смысле производительности программы нет никакой разницы в обеих формах обращения к членам объекта документа. Но в смысле читабельности текста программы второй вариант, несомненно, имеет преимущество. Встроенные функции расширяются в скомпилированной программе так же, как и макросы, но в отличие от мак-

росов компилятор при работе со встроенными функциями выполняет проверку типов аргументов.

Обратите внимание на то, что как класс представления, так и класс документа перегружают часть функций-членов своих базовых классов. Функция OnDraw() является одной из важнейших среди всех виртуальных функций, она является именно тем инструментом, с помощью которого производится рисование в окне приложения. Что касается других функций, то MFC вызывает PreCreateWindow() перед тем, как создается и присоединяется к объекту класса окон MFC окно Windows. Эта функция дает возможность модифицировать такие атрибуты окна, как положение и размер. Функция OnPreparePrinting() используется для модификации диалогового окна Print перед его выводом на экран. Функция OnBeginPrinting() дает шанс создать GDI-объект, такой как кисть или перо, который необходим для выполнения некоторых задач в процессе печати. И, наконец, в функции OnEndPrinting() можно уничтожить любой объект, созданный функцией OnBeginPrinting().

Существует разница между экземпляром класса окна MFC и элементом Windows, который он представляет. Например, когда вы создаете объект класса фрейма MFC, в действительности создаются два объекта – MFC-объект класса окна, который имеет члены-функции и переменные, и окно Windows, которым можно манипулировать, используя функции MFC-объекта. Элемент Windows ассоциируется с соответствующим классом, но сам по себе также представляет некую сущность.

В классе документа за перерисовку окна отвечает функция UpdateAllViews(), в классе представления – функция Invalidate().

Вызов функции Invalidate() приводит к тому, что MFC обращается к функции OnDraw(), которая, в свою очередь, перерисовывает изображение на экране. Invalidate() принимает единственный аргумент (его значение по умолчанию – TRUE), который указывает, нужно ли стирать прежнее изображение перед прорисовкой нового функцией OnDraw(). Бывают случаи, когда удобнее вызывать Invalidate() с аргументом FALSE, и тогда OnDraw() будет рисовать поверх прежнего изображения.

4.2. Создание приложения с однодокументным интерфейсом

При разработке программ в VC++ и MFC обычно предполагается использование архитектуры «документ/представление». Названная архитектура позволяет связать данные с их представлением пользователю на экране. Логическое разделение данных программы и методов их визуального представления позволяет отображать документы разными способами, связав документ с несколькими представлениями (например в Microsoft Word доступны несколько представлений одного и того же документа, они представлены в меню Вид). Изменения, вносимые в документ в одном представлении, отображаются во всех других. В разработке приложений можно использовать как

готовые представления (основанные на элементах управления, таких как деревья просмотра, списки), так и создавать собственные, перегружая функцию отображения, обработчики сообщений от клавиатуры, мыши и пунктов меню.

Архитектура «документ/представление» предоставляет множество возможностей для работы с документом. Так, AppWizard способен генерировать каркас приложения, реализующий документы и представления средствами классов, производных от классов CDocument и CView (классы документа и представления).

Класс документа в MFC отвечает за хранение данных, а также за их загрузку из файлов на диске; содержит функции, позволяющие другим классам (в частности классу представления) получать или изменять данные таким образом, чтобы они были доступны для просмотра и редактирования. Этот класс должен обрабатывать команды меню, непосредственно воздействующие на данные документа.

Представление – это часть программы, использующая библиотеку MFC для управления окном просмотра, обработки информации, вводимой пользователем, и отображения документа в окне.

Пример создания однодокументного приложения рисования в окне прямых линий представлен в [1].

5. Сообщения и команды

5.1. Обработка сообщений

Если и существует некоторая особенность, отличающая программирование в Windows от других областей программирования, то это сообщения. Сообщения являются тем средством, с помощью которого операционная система может дать знать приложению, что что-то произошло, например пользователь нажал клавишу на клавиатуре, или щелкнул кнопкой мыши, или передвинул мышь, или подготовил принтер к выводу информации. Окно, а каждый информационный элемент на экране есть своего рода окно, также может посылать сообщения другому окну, и, как правило, большинство окон реагирует на полученное сообщение тем, что пересылает его дальше, третьему окну, слегка видоизменив. Значительную помощь в организации работы с сообщениями оказывает MFC, скрывая от программиста многие подробности процесса.

Хотя операционная система и использует целые числа для идентификации событий, в тексте программы мы будем иметь дело с символьными идентификаторами. Огромное количество директив #define связывает символьные идентификаторы с соответствующими числами. Префикс WM_ означает Window Message (сообщение Windows).

```
#define WM_SETFOCUS           0x0007
#define WM_KILLFOCUS          0x0008
#define WM_ENABLE             0x000A
#define WM_SETREDRAW          0x000B
#define WM_SETTEXT            0x000C
#define WM_GETTEXT            0x000D
#define WM_GETTEXTLENGTH      0x000E
#define WM_PAINT              0x000F
#define WM_CLOSE              0x0010
#define WM_QUERYENDSESSION    0x0011
#define WM_QUIT               0x0012
#define WM_QUERYOPEN          0x0013
#define WM_ERASEBKGND         0x0014
#define WM_SYSCOLORCHANGE     0x0015
#define WM_ENDSESSION         0x0016
```

Сообщению известно, для какого окна оно предназначено. Оно может иметь до двух параметров. Часто в эти два параметра упаковывается несколько совершенно различных величин.

Обработка разных сообщений выполняется разными компонентами операционной системы и приложения. Например, когда пользователь передвигает мышь по полю окна, формируется сообщение WM_MOUSEMOVE, которое передается окну, а окно, в свою очередь, передает это сообщение операционной системе. И уже последняя перерисовывает указатель мыши в

новом месте. Когда пользователь щелкает левой кнопкой мыши на экранной кнопке, кнопка, которая также есть особый вид окна, получает сообщение WM_LBUTTONDOWN. В процессе обработки этого сообщения кнопка часто формирует новое сообщение для окна, в котором она находится, причем это сообщение гласит: «Ой, на мне щелкнули!».

Библиотека MFC позволяет программистам в подавляющем большинстве случаев полностью отстраниться от сообщений нижнего уровня, таких как WM_MOUSEMOVE и WM_LBUTTONOOWN. Программист может полностью сосредоточиться на сообщениях более высокого уровня, таких как «Выбран третий элемент такого-то списка» или «Произошел щелчок на кнопке Move». Поступают такого рода сообщения в те программы, которые пишет программист, и в компоненты операционной системы точно так же, как и сообщения нижнего уровня. Единственная разница в том, что MFC берет на себя значительную часть работы по обработке сообщений низкого уровня и позволяет заметно облегчить распределение сообщений между разными классами объектов, на уровне которых и будет производиться их обработка.

5.2. Циклы обработки сообщений

Сердцем любой Windows-программы является цикл обработки сообщений (Message Loop), который практически всегда находится в функции WinMain(). Эта функция в Windows-приложениях играет ту же роль, что и функция Main) в DOS-приложениях, – ее вызывает операционная система сразу же после загрузки приложения в память. Текст типичной функции WinMain() представлен в листинге 5.1.

Листинг 5.1. Текст типичной функции WinMain().

```
int APIENTRY WinMain(HINSTANCE hInstance,
HINSTANCE hPrevInstance,
LPSTR IpCmdLine,
int nCmdShow) {
MSG   msg;
if(!InitApplication(hInstance))
return (FALSE);
if(!InitInstance( hInstance,nCmdShow))
return (FALSE);
while(GetMessage(&msg, NULL, 0, 0)) {
TranslateMessage( &msg);
DispatchMessage( &msg):
}
return (msg.wParam);
}
```

В Си-программах для Windows функция InitApplication() вызывает RegisterWindow(), а InitInstance() – CreateWindow(). Затем наступает очередь цикла обработки сообщений. Он представляет собой типичную циклическую

конструкцию Си на базе оператора while, внутри которой вызывается функция GetMessage(). Эта функция API заполняет msg кодом сообщения, которое операционная система распределила для этого приложения, и почти всегда возвращает TRUE. Таким образом цикл повторяется снова и снова до тех пор, пока работает приложение. Единственный вариант, при котором GetMessage() возвращает FALSE, – получение сообщения WM_QUIT.

При работе с сообщениями, поступающими с клавиатуры, некоторую часть предварительной обработки берет на себя функция API TranslateMessage(). Ее назначение состоит в следующем. Прикладной части программы нет дела до сообщений наподобие «Нажата клавиша <A>» и «Отпущена клавиша <A>». Прикладную часть интересует только то, какую литеру (символ) ввел пользователь, т. е. ее вполне удовлетворит сообщение «Введен символ A». Вот это преобразование – нескольких сообщений о деталях процесса в одно сообщение о его сути – и выполняет функция TranslateMessage(). Она перехватывает сообщения WM_KEYDOWN и WM_KEYUP и вместо них посылает сообщение WM_CHAR. Пользователь вводит текст в текстовое поле или в другой элемент управления, и забота программиста – извлечь введенный текст из этого объекта после того, как пользователь щелкнет на OK.

Функция API DispatchMessage() вызывает, в свою очередь, функцию WndProc() того окна, для которого предназначено сообщение. Типичная функция WndProc() в Си-программе для Windows представляет собой огромный оператор switch с отдельными case для каждого сообщения, которое приложение намеревается самостоятельно обрабатывать. Текст ее приведен в листинге 5.2.

Листинг 5.2. Текст типичной функции WndProc().

```
LONG APIENTRY MainWndProc(HWND hwnd,  // Дескриптор окна.
UINT msg,       //Тип сообщения.
UINT wParam,    //' Дополнительная информация.
LONG lParam)    //Дополнительная информация.
{
switch(msg) {
case WM_MOUSEMOVE:
// Обработка перемещения мыши.
break;

case WM_LBUTTONDOWN:
// Обработка щелчка левой кнопки мыши.
break;
case WM_ RBUTTONDOWN:
// Обработка щелчка правой кнопки мыши.
break;
case WM_PAINT :
// Перерисовать окно.
break;
```

```
case WM_DESTROY :    // Сообщение: окно будет уничтожено.
PostQuitMessage( 0);
return 0;
break;
default:
return (DefWindowProc(hwnd, msg, wParam, lParam));
}
return (0);
}
```

Подобная функция очень длинна в более или менее порядочном приложении. MFC решает проблему следующим образом: информация о сообщениях, которые должны обрабатываться, расположена в карте сообщений класса. Таким образом, отпадает необходимость в огромных операторах switch, в которых сосредоточено распределение сообщений.

5.3. Карты сообщений

Использование карты сообщений (Message maps) лежит в основе подхода, который реализуется в MFC для программирования Windows-приложений. Суть его состоит в том, что от разработчика требуется только написать функции обработки сообщений и включить в свой класс карту сообщений. После этого операционная система будет отвечать за то, чтобы сообщение было передано именно той функции, которая будет его обрабатывать.

Карта сообщений состоит из двух частей: одна – в файле заголовка для класса .h, а другая – в соответствующем файле реализации .cpp. Они, как правило, формируются мастерами, хотя в некоторых случаях можно сделать это (или частично отредактировать их) и самостоятельно. В листинге 5.3 представлена часть текста файла заголовка одного из классов простого приложения работы с базами данных Employee.

Листинг 5.3. Фрагмент карты сообщений класса CEmployeeView в файле заголовка CEmployeeView.h.

```
//{{AFX_MSG(CEmployeeView)
    afx_msg void OnRecordAdd();
    afx_msg void OnRecordDelete();
    afx_msg void OnSortId();
    afx_msg void OnSortName();
    afx_msg void OnSortRate();
    afx_msg void OnSortDepartment();
    afx_msg void OnFilterDepartment();
    afx_msg void OnFilterId();
    afx_msg void OnFilterName();
    afx_msg void OnFilterRate();
    afx_msg void OnFileSaveAs();
    afx_msg void OnRecordFind();
```

```
//}}AFX_MSG
     DECLARE_MESSAGE_MAP()
```

Специальным образом оформленный комментарий позволяет ClassWizard определить, какие именно сообщения перехватываются этим классом. DECLARE_MESSAGE_MAP – это макрос, расширяемый препроцессором компилятора Visual C++, в котором объявляются переменные и функции, принимающие участие в организации перехвата сообщений. Карта сообщений в файле EmployeeView.cpp представлена в листинге 5.4.

Листинг 5.4. Фрагмент карты сообщений класса CEmployeeView в файле реализации CEmployeeView.cpp.

```
BEGIN_MESSAGE_MAP(CEmployeeView, CRecordView)
//{{AFX_MSG_MAP(CEmployeeView)
     ON_COMMAND(ID_RECORD_ADD, OnRecordAdd)
     ON_COMMAND(ID_RECORD_DELETE, OnRecordDelete)
     ON_COMMAND(ID_SORT_ID, OnSortId)
     ON_COMMAND(ID_SORT_NAME, OnSortName)
     ON_COMMAND(ID_SORT_RATE, OnSortRate)
     ON_COMMAND(ID_SORT_DEPARTMENT, OnSortDepartment)
     ON_COMMAND(ID_FILTER_DEPARTMENT, OnFilterDepartment)
     ON_COMMAND(ID_FILTER_ID, OnFilterId)
     ON_COMMAND(ID_FILTER_NAME, OnFilterName)
     ON_COMMAND(ID_FILTER_RATE, OnFilterRate)
     ON_COMMAND(ID_FILE_SAVE_AS, OnFileSaveAs)
     ON_COMMAND(ID_RECORD_FIND, OnRecordFind)
     //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

Макросы BEGIN_MESSAGE_MAP и END_MESSAGE_MAP, так же как DECLARE_MESSAGE_MAP в файле заголовка, объявляют члены (переменные и функции), которые программа должна использовать для того, чтобы разобраться в картах всех объектов системы. Существует довольно большой набор макросов, используемых для работы с картой сообщений. Некоторые из них перечислены ниже:

– DECLARE_MESSAGE_MAP. Используется в файле заголовка для того, чтобы объявить, что в файл собственно текста программы будет включена карта сообщений;

– BEGIN_MESSAGE_MAP. Отмечает начало карты сообщений в тексте программы;

– END_MESSAGE_MAP. Отмечает конец карты сообщений в тексте программы;

– ON_COMMAND. Используется для того, чтобы перенаправить обработку некоторой команды функции-члену класса.

– ON_CONTROL. Используется для того, чтобы перенаправить обработку кода извещения от элемента управления функции-члену класса;

– ON_MESSAGE. Используется для того, чтобы перенаправить обработку некоторого сообщения функции-члену класса;

– ON_UPDATE_COMMAND_UI. Используется для того, чтобы перенаправить обновление, связанное с заданной командой, функции-члену класса;

– ON_NOTIFY. Используется для того, чтобы перенаправить функции-члену класса обработку заданного кода извещения, который сопровождается дополнительными данными от элемента управления.

Обработка происходит следующим образом. Функции, вызванные циклом обработки сообщений, решают на основании этой таблицы, какой из объектов и какая из функций-членов этого объекта будет обрабатывать сообщение.

5.4. Организация обработки сообщений в приложении

Преимущество MFC состоит в том, что в этой библиотеке уже имеются готовые классы, которые перехватывают и обрабатывают большинство распространенных сообщений, причем делают это безо всяких усилий со стороны разработчика программы. Например, не нужно заботиться об обработке таких сообщений, как вызов команды File→Save As. Классы MFC самостоятельно «отловят» это сообщение, выведут на экран диалоговое окно для ввода нового имени файла, обработают все манипуляции пользователя в этом окне, сделают всю черновую работу и в конце вызовут функцию Serialize() сериализуемого объекта, которая запишет данные в файл. AppWizard, как правило, формирует пустую функцию Serialize(), в которую разработчик должен вставить необходимый текст. Таким образом, программисту необходимо вставлять в карту сообщений компоненты только для тех случаев, когда обработка некоторого сообщения в данном приложении отличается от общепринятой методики.

Формирование карты сообщений с помощью ClassWizard рассмотрим на примере обработки сообщения нажатия левой кнопки мыши в окне представления. Выбираем нужный класс (пусть это будет CMiniDrawView), вызываем контекстное меню, в котором выбираем команду Properties (рис. 5.1). В правой части окна появится окно Properties (рис. 5.2).

В окне Properties выбираем закладку Messages (рис. 5.3). Находим необходимый обработчик (в нашем случае WM_LBUTTONDOWN) и в выплывающем списке выбираем команду Add OnLButtonDown (рис. 5.4).

В результате шаблон обработчика сообщения с именем OnLButtonDown добавлен в определение класса CMiniDrawView в файле MiniDrawView.h, реализация функции добавлена в файл MiniDrawView.cpp. Добавлена функция OnLButtonDown в схему сообщений класса. Далее следует редактировать код этой функции.

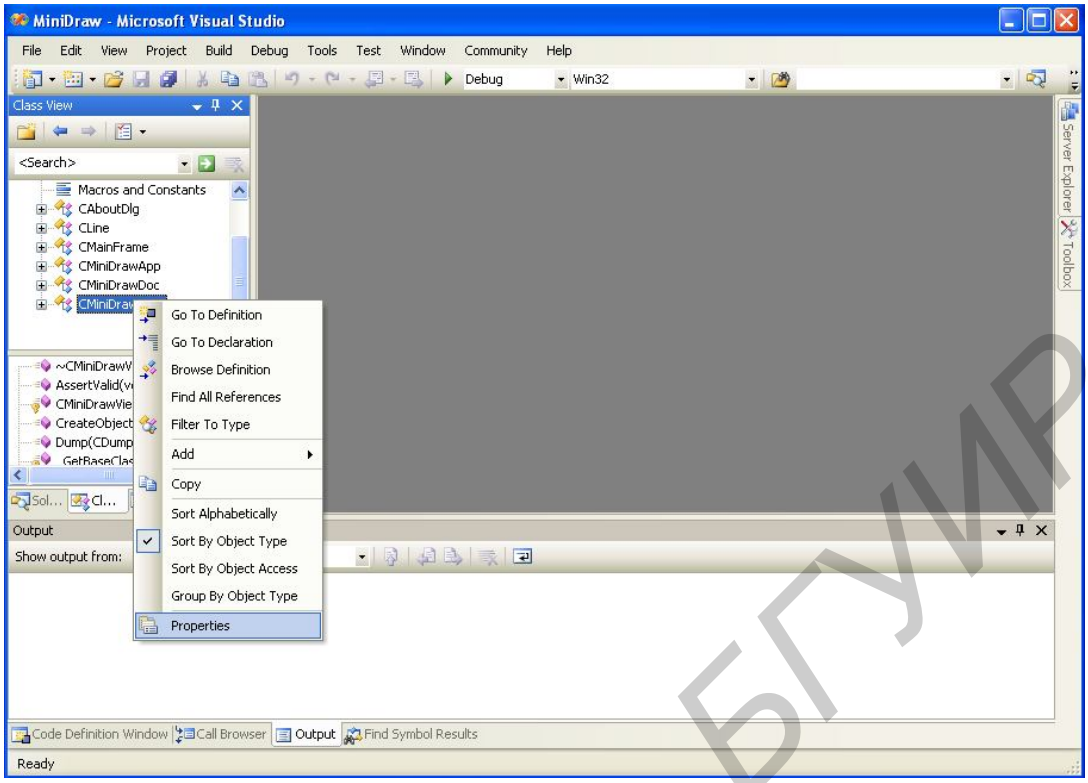

Рис. 5.1. Выбор команды Properties в контекстном меню

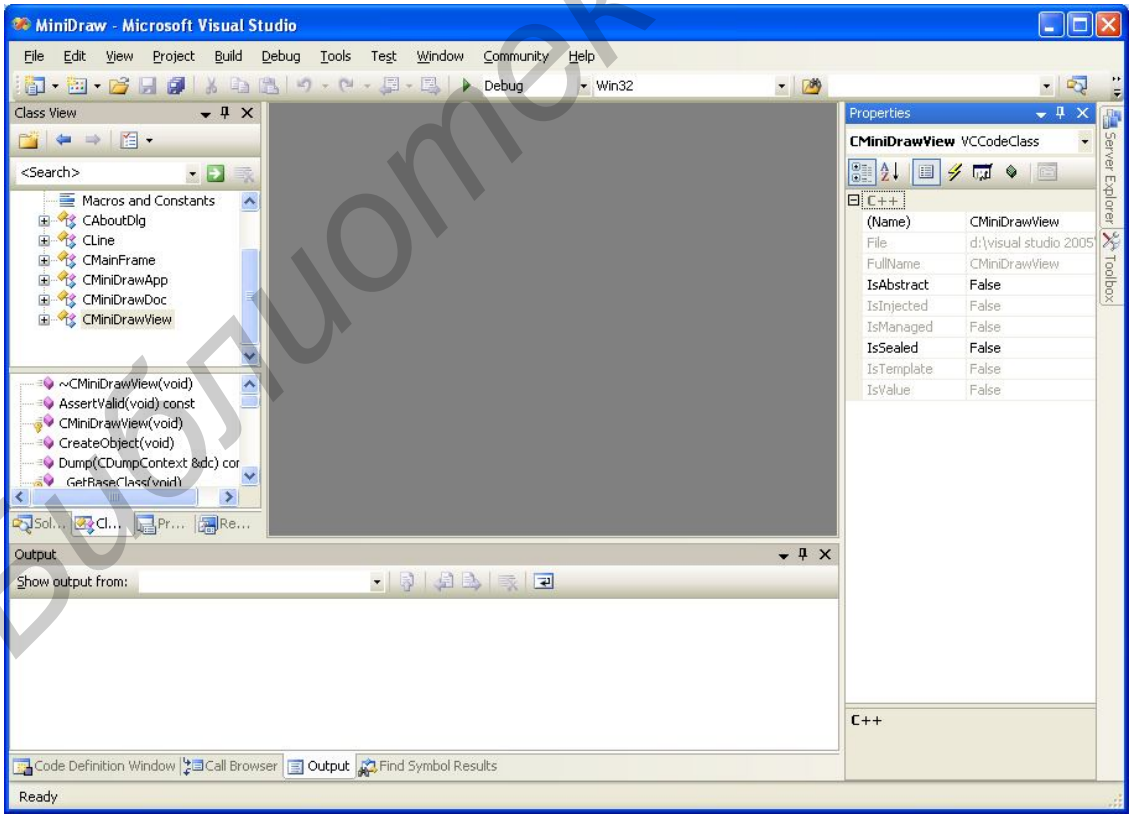

Рис. 5.2. Окно Properties

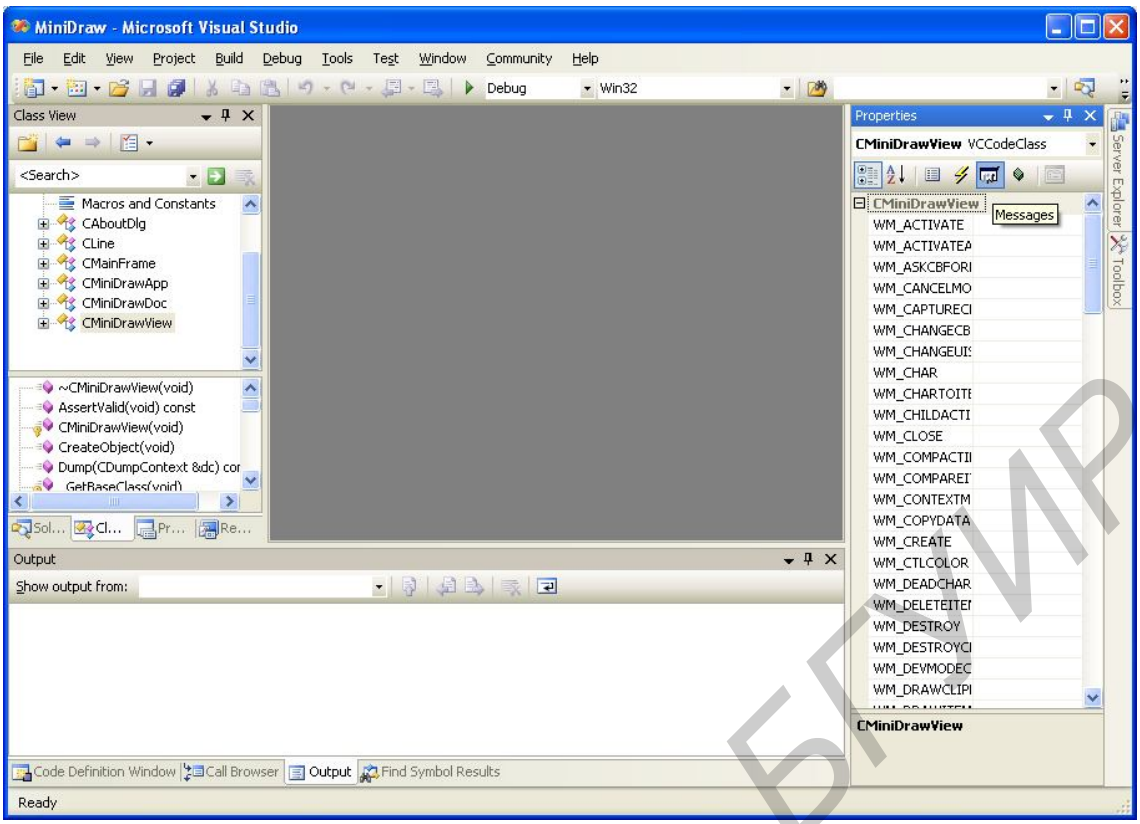

Рис 5.3. Выбор списка «Messages»

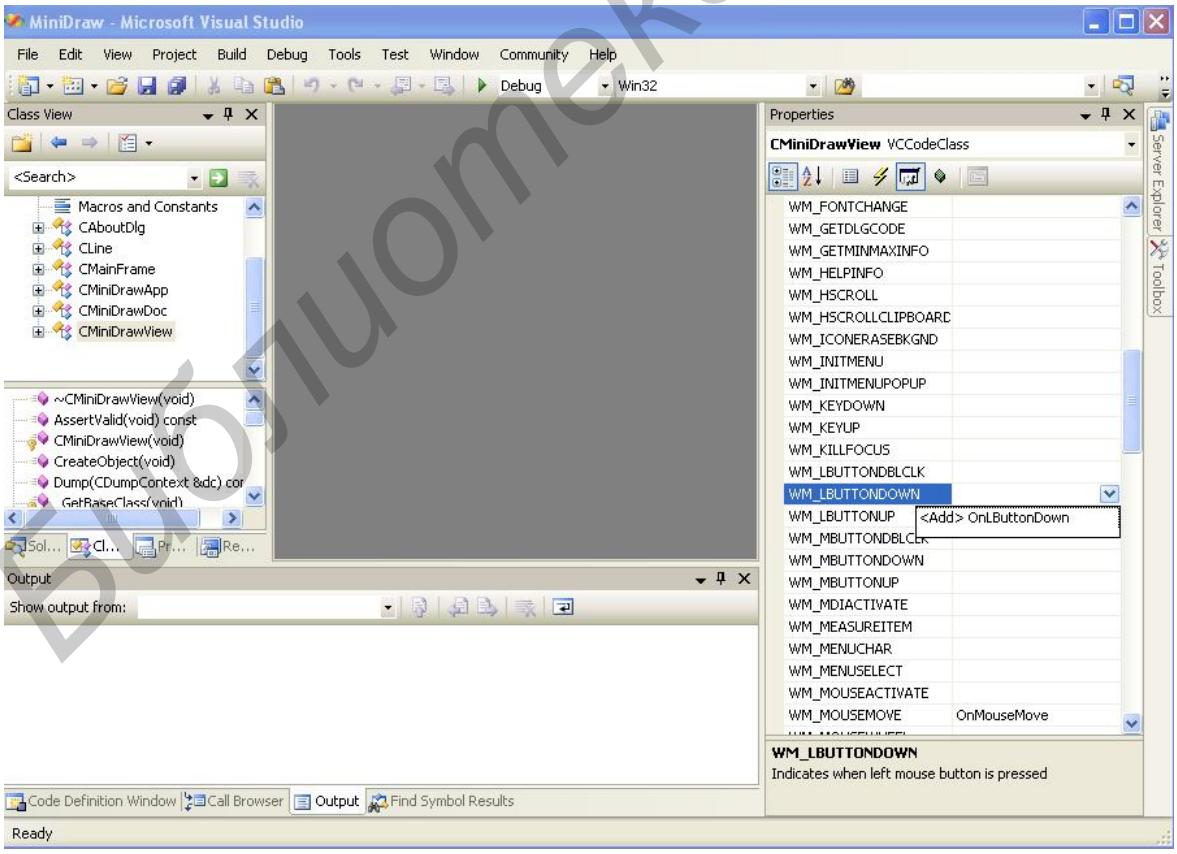

Рис. 5.4. Добавление обработчика для сообщения WM_LBUTTONDOWN

5.5. Команды

Команда – это сообщение специального типа, которое формируется в тех случаях, когда пользователь выбирает пункт меню, щелкает на кнопке или каким-либо другим способом дает системе понять, что ему что-то от нее нужно. Выбор из меню порождает сообщение WM_COMMAND, а щелчок на кнопке или выбор в списке – сообщение WM_NOTIFY с кодом извещения от элемента управления.

Все сообщения команд содержат в качестве первого параметра идентификатор ресурса выбранного пункта меню или кнопки. Этот идентификатор ресурса присваивается в соответствии со стандартом на форматы подобного рода идентификаторов, например для пункта Save As меню File идентификатор будет иметь вид ID_FILE_SAVE.

Получать сообщения могут только объекты классов-наследников CWnd, а все объекты классов, порожденных от CCmdTarget, включая CWnd и CDocument, могут получать команды и извещения. Это означает, что класс, который наследует CDocument, может иметь карту сообщений, причем в ней не должно быть ни одного компонента, соответствующего сообщению, а только компоненты для команд и извещений. Тем не менее этот фрагмент программы по-прежнему называется картой сообщений.

Следует позаботиться о правильном выборе классов, которые будут обрабатывать все события, которые могут произойти в разрабатываемом приложении. Если пользователь изменяет размеры окна, посылается сообщение WM_SIZE, и вам, возможно, понадобится изменить масштаб изображения или выполнить еще что-нибудь с представлением в приложении. Если пользователь выбирает некоторый пункт меню, формируется команда, а это означает, что класс документа должен что-то сделать в ответ на нее.

5.5.1. Обновление команд

Рассмотрим, как программа выполняет блокировку определенных пунктов меню или кнопок в соответствии с контекстом задачи. Этот процесс назван обновлением команд (command updating). В процессе функционирования приложения может возникнуть необходимость блокировать некоторые команды меню, чтобы показать, что они в данный момент недоступны. Реализовать эту идею можно двумя способами.

Один состоит в том, чтобы организовать огромную таблицу, элементами которой будут все имеющиеся в приложении пункты меню, каждому из которых сопоставлен флаг. Состояние флага (TRUE или FALSE) указывает, доступен ли этот пункт меню. Как только возникает необходимость вывести меню на экран, можно просмотреть таблицу, и все сразу станет ясно. При любой операции, которая может повлечь за собой изменения в статусе какого-либо пункта меню, таблица обновляется. Все это в совокупности называется подходом непрерывного обновления (continuous-updating approach).

Другой подход состоит в том, чтобы, не имея такой таблицы, перед каждым выводом меню на экран анализировать все условия, которые влияют на возможную блокировку. Он называется подходом обновления по требованию (update-on-demand approach). Именно такой подход и реализован в Windows.

Когда наступает время выводить на экран меню, конкретные объекты «знают», нужно ли блокировать связанный с ними пункт меню. Например, объект класса документа знает, был ли он модифицирован после последнего сохранения, и решает, стоит ли блокировать пункт Save меню File. Объект класса представления знает, есть ли выделенный фрагмент текста, и может решить, как поступить с пунктами Cut и Copy меню Edit. Все это означает, что комплексная задача блокировки пунктов меню в соответствии с контекстом приложения распределяется между различными объектами приложения, а не возлагается на главную вызывающую подпрограмму WndProc().

Подход, реализованный в MFC, состоит в том, чтобы использовать объект класса CCmdUI (класс интерфейса с командами пользователя – command user interface) и предоставить ему возможность перехватывать любые сообщения ON_UPDATE_COMMAND_UI. Организовать такой перехват можно, добавив (или предоставив возможность ClassWizard добавить) макрос ON_UPDATE_COMMAND_UI в карту сообщений. Объект класса CCmdUI также используется для блокировки или разблокировки командных кнопок и других элементов управления.

Класс CCmdUI имеет следующие функции-члены:

Enable() – принимает аргумент TRUE или FALSE. Блокирует элемент интерфейса пользователя, если передан аргумент FALSE, в противном случае делает элемент доступным;

SetCheck() – устанавливает состояние радиопереключателя или флажка (как выбранное, невыбранное или неопределенное);

SetRadio() – включает или выключает элемент управления как принадлежащий к группе зависимых переключателей, из которых только один может быть включен;

SetText() – устанавливает текст надписи пункта меню или кнопки.

5.5.2. Обработка командных сообщений

Рассмотрим добавление в класс документа CMiniDrawDoc проекта MiniDraw обработчика командного сообщения для пункта меню DeleteAll меню Edit.

1. Выбираем закладку Class View в окне проекта.

2. Находим класс CMiniDrawDoc и вызываем контекстное меню, в котором выбираем команду Properties.

3. В окне Properties выбираем закладку Events (рис. 5.5).

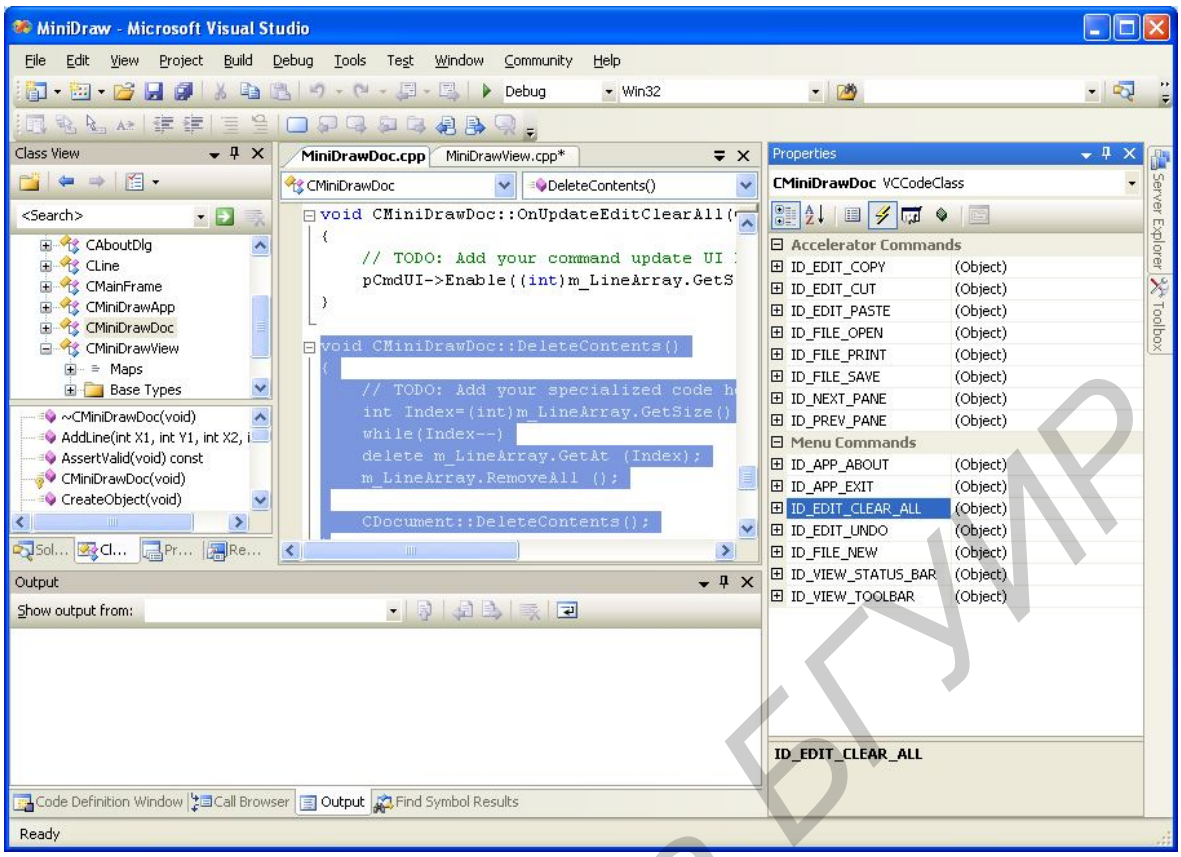

Рис. 5.5. Открытие закладки Events окна свойств Properties

4. В списке выбираем идентификатор ID_EDIT_CLEAR_ALL.

5. В выпадающем списке отобразятся идентификаторы двух типов сообщений (рис. 5.6), которые эта команда может передавать объекту класса документа: COMMAND и UPDATE_COMMAND_UI. Идентификатор COMMAND указывает на сообщение, передаваемое при выборе пользователем пункта меню. Идентификатор UPDATE_COMMAND_UI указывает на сообщение, передаваемое при первом открытии меню, содержащего команду.

6. Выбирам команду OnEditClearAll справа от идентификатора COMMAND, после чего добавляется объявление функции в класс CMiniDrawDoc в файле MiniDrawDoc.h и минимальное определение функции в файл MiniDrawDoc.cpp. Генерируется код для добавления этого обработчика в схему обработки сообщений класса документа CMiniDrawDoc.

7. Справа от идентификатора обработчика в выплывающем списке выбираем команду Edit Code, после чего откроется файл MiniDrawDoc.cpp и отобразит только что созданную функцию OnEditClearAll, чтобы можно было добавить в нее код.

Сообщение UPDATE_COMMAND_UI посылается при первом открытии меню, содержащего команду Delete All. Сообщение посылается до того, как меню станет видимым, обработчик может использоваться для инициализации команды в соответствии с текущим состоянием программы.

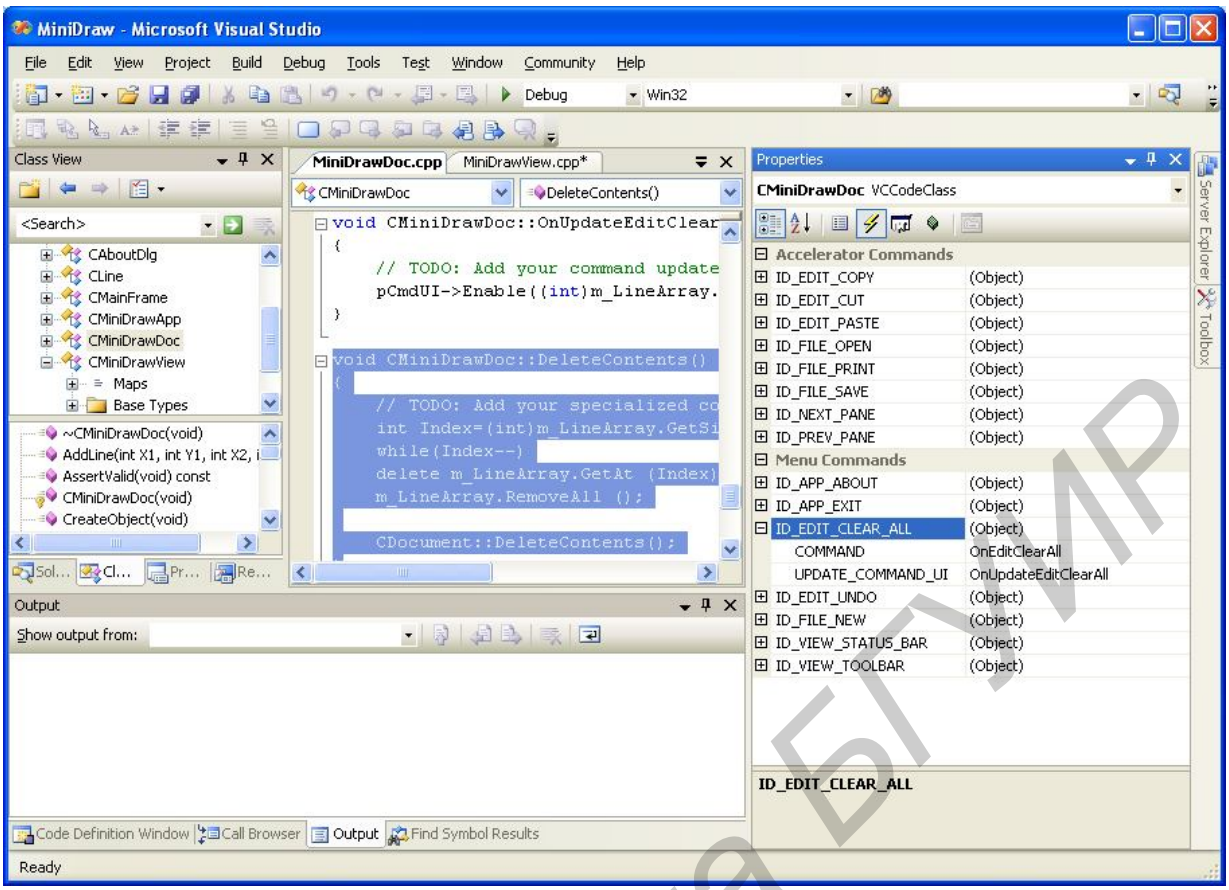

Рис. 5.6. Выбор обработчика для команды Delete All

5.6. Обработка сообщений от панели инструментов

Создание хорошего пользовательского интерфейса – это едва ли не половина успеха в разработке приложения для Windows. Visual C++ и его мастера предоставляют разработчику уникальные возможности для создания приложений, поддерживающих все привычные элементы пользовательского интерфейса, включая меню, диалоговые окна, панели инструментов и строку состояния.

Пиктограммы на панелях инструментов так же, как и элементы меню, соответствуют определенным командам. Хотя создать панель инструментов в приложении можно с помощью AppWizard, все же потребуется написать программный код для окончательной доработки. С помощью AppWizard можно создать только стандартную панель инструментов, в которую будут включены пиктограммы, встречающиеся в большинстве приложений. При разработке собственной панели инструментов для отражения специфики набора команд потребуется вставить новые или удалить существующие пиктограммы. Стандартная панель инструментов содержит пиктограммы для наиболее распространенных команд меню File, Edit, View, Format, Window, а также пиктограмму для отображения окна About. Пример работы с пиктограммами панели инструментов представлен в лабораторной работе №4.

6. Вывод на экран графической информации

Большинство приложений нуждается в выводе информации на экран. Windows является системой, не зависящей от аппаратных средств, поэтому большая часть нагрузки по выводу информации перекладывается на программиста. Заранее неизвестно, с устройством какого типа придется иметь дело приложению, но необходимо задать все необходимые параметры для его настройки. Средства вывода работают с аппаратурой через промежуточное звено, которое называется контекстом устройства (device context – DC).

Независимость Windows от аппаратных средств, с одной стороны, усложняет работу программиста в связи с необходимостью программирования операций отображения информации; с другой стороны, упрощает настройку программы на каждый новый вариант аппаратуры отображения. В большинстве случаев Windows управляет устройствами посредством специальных программ – драйверов. Драйверы принимают информацию от приложения и передают данные конкретному устройству – монитору, принтеру или какому-нибудь другому устройству. Драйвер связывается с приложением посредством контекста устройства.

Контекст устройства – это структура C++, которая содержит атрибуты рабочего поля окна. Атрибуты включают выбранное для текущей операции перо, кисть и шрифт. Контекст устройства в каждый момент времени располагает только одним пером, кистью или шрифтом. Если необходимо некоторую часть изображения нарисовать другим пером, например более толстым, придется создать новое перо, внести его в контекст устройства вместо старого. Если необходимо заливать контуры красной кистью, придется ее создать и «выбрать в контекст» – так программисты называют операцию замены инструмента в контексте устройства.

Рабочая область окна (window's client area) – это часть поверхности экрана, в которой можно отображать все, что посчитает нужным приложение: текст, таблицы данных, картинку и т. д. Библиотека MFC инкапсулирует функции графического интерфейса Windows (Graphic Device Interface – GDI) в свои классы контекста устройства. Контекст устройства можно представить как лист бумаги, на котором будет производиться рисование. Драйвер устройства – это специальная программа, которая умеет контекст перевести на конкретное устройство.

В MFC есть несколько контекстов устройств и все наследники от CDC: CClientDC, CWindowDC, CMetaFileDC, CPaintDC. Каждый контекст предназначен для рисования в определенной области. При этом CPaintDC и CClientDC очень похожи. Оба эти класса предназначены для рисования внутри клиентской области окна. То есть они не могут добраться до меню или рамки окна. CPaintDC используется только тогда, когда отвечает на сообщение WM_PAINT. Вместе с этим сообщением будет передана та область, которая требует перерисовки, т. к. окно может быть на экране не полностью.

Для извещения о перерисовке вызываются функции BeginPaint и EndPaint, без их вызова Windows будет считать, что перерисовка не произошла. В этом заключается существенное отличие CPaint, в его конструкторе автоматически вызывается BeginPaint, а в деструкторе – EndPaint. CClientDC не выполняет эти функции автоматически. Конструктор CClientDC вызывает функцию GetDC, а деструктор – ReleaseDC. Класс CWindowDC позволяет рисовать поверх меню и везде в пределах рамки окна. Контекст CMetaFileDC используется для работы с метафайлами, рисование из которых может выполняться многократно.

Функции, находящиеся в классе контекста устройства, обеспечивают:

– связь с физическим устройством;
– набор изобразительных средств;
– регулирование вывода.

Смысл физической связи в том, что можно рисовать на устройстве, не заботясь о том, как это устройство физически работает. Например, программиста не интересует производитель, марка и другие особенности монитора, принтера или другого устройства. За это отвечает драйвер этого устройства. Набор изобразительных средств – это то, чем вы можете рисовать. Контекст устройства дает вам перо, кисть, умеет делать некоторые операции по рисованию примитивов. То есть он дает инструментарий по рисованию. Регулировка вывода связана с необходимостью следить за тем, что часть экрана может перекрываться другими областями, и поэтому в них рисовать нельзя.

6.1. Классы изобразительных средств и рисование простейших фигур

В контексте устройства есть ряд классов изобразительных средств, которые являются наследниками CGdiObject. В MFC их шесть: CBitmap, CBrush, CFont, CPalette, CPen, CRgn. CBitmap – это класс, который умеет работать с растровыми изображениями. CBrush – это кисть для рисования. CFont – класс шрифта текста. CPalette – класс, умеющий работать с цветовыми палитрами. CPen – класс пера и CRgn – класс региона, т. е. области вывода.

В листинге 6.1 представлены примеры рисования простейших фигур с комментариями.

Листинг 6.1. Примеры рисования простейших фигур.

```
//Вывести строку:
void CMainWnd::OnPaint()
{    CPaintDC dc(this);
     dc.TextOut( X1, Y1,"TextOut Samples");
}
//Вывести точку, чем выше разрешение экрана, тем меньше точка.
void CMainWnd::OnPaint()
{    CPaintDC dc(this);
```

```
     dc.SetPixel(500,200,RGB(255,0,0));
}
//Дуга окружности:
void CMainWnd::OnPaint()
{    CPaintDC dc(this);
     dc.Arc(200,200,100,100,400,400,10,10);
}
//Замкнутая дуга:
void CMainWnd::OnPaint()
{    CPaintDC dc(this);
     dc.Chord(250,250,100,100,400,400,10,10);
}
//Эллипс:
void CMainWnd::OnPaint()
{    CPaintDC dc(this);
     dc.Ellipse(450,450,50,150);
}
//Линия:
void CMainWnd::OnPaint()
{    CPaintDC dc(this);
     dc.MoveTo(200,200);
     dc.LineTo(100,100);
}
```

6.2. Изменение размеров и положения окна

В программе, использующей MFC, можно изменить размеры и положение окна с помощью функции PreCreateWindow() – члена класса главного окна приложения CMainFrame. Она вызывается автоматически перед началом формирования главного окна приложения. В главном окне содержатся все видимые объекты приложения и определяется размер представления.

Функция PreCreateWindow() имеет один аргумент – ссылку на экземпляр структуры CREATESTRUCT. Эта структура содержит всю информацию об окне, которое должно появиться на экране. Особое внимание тех, кто программирует с помощью MFC, привлекают члены cx, cy, x и y. Изменяя cx и cy, можно регулировать ширину и высоту окна, а изменяя x и y, – положение окна на экране. Перегрузив функцию PreCreateWindow(), можно изменить структуру CREATESTRUCT до того, как Windows использует ее для формирования окна.

Пример переопределения функции PreCreateWindow() представлен в листинге 6.2. Операторы устанавливают новые высоту и ширину окна приложения.

Листинг 6.2. Пример переопределения функции PreCreateWindow().

```
BOOL CMainFrame:: PreCreateWindow(CREATESTRUCT& cs)
{
cs.cx = 440;
cs.cy = 480;
```

```
if(!CFrameWnd::PreCreateWindow(cs)) return FALSE;
return TRUE;
}
```

6.3. Как выполняется рисование в программе, использующей MFC

Одно из основных сообщений, которое должна уметь обрабатывать любая Windows-программа, – сообщение WM_PAINT. Операционная система Windows посылает это сообщение окну приложения при любой операции, требующей перерисовки изображения в окне. Несколько событий могут стать причиной возникновения необходимости в такой перерисовке.

Первое из них – запуск программы. В правильно организованной Windows-программе окно приложения получает сообщение WM_PAINT практически немедленно после запуска, с тем чтобы данные, соответствующие исходному состоянию приложения, сразу же были предъявлены пользователю. Другое событие, требующее перерисовки окна, а значит, и генерирующее сообщение WM_PAINT, – изменение размеров окна или перекомпоновка окон на экране. В последнем случае окно приложения может стать верхним или быть частично перекрытым другим окном. В любом случае приоткрывается хотя бы часть окна, и изображение в нем должно быть перерисовано. И, наконец, программа может посылать сообщение WM_PAINT сама себе, с тем чтобы удалить старые данные на экране и вывести новые. Такая возможность позволяет приложению всегда держать пользователя в курсе происходящих событий.

В карте сообщений класса макрос для сообщения WM_PAINT будет называться ON_WM_PAINT(), а соответствующая функция – OnPaint().

Текст функции CView::OnPaint() представлен ниже.

```
void CView::OnPaint()
{
// Стандартная последовательность вызовов для прорисовки.
CPaintDC dc(this);
OnPrepareDC(&dc);
OnDraw(&dc);
}
```

Класс CPaintDC – это специальный класс для управления контекстами устройств, которые используются только для реакции на сообщения WM_PAINT. Объект класса CPaintDC делает нечто большее, чем просто создание контекста устройства. Он вызывает функцию BeginPaint() в конструкторе класса и функцию EndPaint() в деструкторе. В процессе реакции на сообщение WM_PAINT необходимо вызывать и BeginPaint(), и EndPaint(). Конструктор класса CPaintDC требует одного аргумента – указателя на объект, представляющий окно, для которого и создается контекст устройства. Указатель this указывает на текущее окно, т. е. конструктору дается задание создать контекст для текущего окна.

Функция OnPrepareDC() является членом класса CView. Она подготавливает контекст устройства для дальнейшего использования. Функция OnDraw() берет на себя всю работу по обновлению представления документа на экране. В большинстве случаев для каждого приложения нужно разрабатывать собственную функцию OnDraw() и никогда не вносить изменений в предлагаемый MFC текст функции OnPaint().

6.4. Использование перьев

Для пера требуется определить стиль начертания линии, ее толщину и цвет. В листинге 6.3 представлена функция ShowPens() , которая формирует различные типы перьев в теле цикла for.

Листинг 6.3.

```
void CPaint1View::ShowPens(CDC * pDC)
{
// Инициализировать положение линии.
UINT position =10;
// Начертить шестнадцать линий.
for (UINT x=0; x<16; ++x)
{
// Сформировать новое перо и выбрать его в контекст.
CPen pen(PS_SOLID, x*2+1, RGB(0, 0, 255));
CPen* oldPen = pDC->SelectObject(&pen);
// Начертить линию новым пером.
position += x * 2 + 10;
pDC->MoveTo(20, position);
pDC->LineTo(400, position);
// Восстановить прежнее перо в контексте.
pDC->SelectObject(oldPen);
}
}
```

В теле цикла ShowPens() сначала формируется новое перо – экземпляр класса CPen. Конструктору требуется передать три параметра. Первый – стиль линии. Варианты стилей представлены в табл. 6.1.

Таблица 6.1

Стили пера

Стиль	Описание
PS_DASH	Перо вычерчивает штриховую линию
PS_DASHDOT	Перо вычерчивает штрихпунктирную линию
PS_DASHDOTOOT	Перо вычерчивает штрихпунктирную линию с двумя точками
PS_ DOT	Перо вычерчивает пунктирную линию
PS_INSIDEFRAME	Перо используется для вычерчивания линий внутри замкнутого контура
PS_NULL	Перо вычерчивает невидимую линию
PS_SOLID	Перо вычерчивает сплошную линию

Только сплошные линии могут иметь регулируемую толщину. Все линии, вычерчиваемые по шаблону, должны иметь толщину 1. Второй параметр – толщина линии, которая увеличивается в каждом последующем цикле. Третий параметр – цвет линии. Макрос RGB принимает три значения соответственно для красной, зеленой и синей составляющих и преобразует их в комбинированный код цвета, воспринимаемый Windows. Значения интенсивностей компонентов находятся в диапазоне 0–255 (естественно, чем больше величина, тем более интенсивный цвет). Пример констант для цветов представлен ниже:

```
RGB(0, 0, 0), // черный
RGB(255,0, 0), // красный
RGB (0,255., 0), // зеленый
RGB(0,0, 255), // синий
RGB(255,255, 0), // желтый
RGB(255,0, 255), // пурпурный
RGB(0,255,255), // голубой
RGB(127,127, 127), // серый
RGB(255,255, 255)};// белый
```

Далее новое перо выбирается в контекст функцией SelectObject(&pen), старое перо сохраняется в переменной oldPen. Метод MoveTo() перемещает перо в точку с координатами X,Y без вычерчивания; метод LineTo() выполняет вычерчивание, «передвигая» перо вдоль прямой линии. При этом используются заказанные стиль, толщина и цвет пера. И последнее – в контексте восстанавливается прежнее перо.

6.5. Работа с кистью

Кисть закрашивает (заливает) внутреннюю область замкнутых фигур. Можно создавать сплошные кисти или стандартные трафаретные. Функция ShowBrushes() (листинг 6.4) продемонстрирует как сплошные кисти, так и стандартные трафаретные, которые будут использованы для заливки прямоугольников.

Листинг 6.4.

```
void CPaintlView::ShowBrushes(CDC * pDC)
{                                    '
// Инициализировать расположение прямоугольника.
UlNT position = 0;
//Выбрать перо для вычерчивания контура прямоугольника.
CPen pen(PS_SOLID, 5, RGB(255, 0, 0));
CPen* oldPen = pDC->SelectObject(&pen);
// Начертить семь прямоугольников.
for (UINT x=0; x<7; ++x)
{
CBrush* brush;
```

```
// Создать сплошную или заштрихованную кисть
if(x==6)
brush = new CBrush(RGB(0,255,0));
else
brush = new CBrush(x, RGB(0,160,0));
//Выбрать новую кисть в контекст.
CBrush* oldBrush = pDC->SelectObject(brush);
// Начертить прямоугольник.
position += 50;
pDC->Rectangle(20, position, 400, position + 40);
// Восстановить контекст и стереть кисть.
pDC->SelectObject(oldBrush);
delete brush;
}
//Восстановить прежнее перо в контексте
pDC->SelectObject(oldPen);
}
```

Все прямоугольники, закрашиваемые различными кистями, будут вычерчены с видимой линией контура. Для этого нужно создать перо (стиль – сплошное, толщина – 5 пикселей, цвет – красное) и выбрать его в контекст устройства. Подобно функции ShowPensO, эта программа использует для демонстрации кистей цикл for. Однако в отличие от предыдущей функции новые объекты кисти создаются вызовом new. Это позволяет использовать то конструктор с одним аргументом, который создает сплошную кисть, то конструктор с двумя аргументами, который создает трафаретную кисть. Первый аргумент двухаргументного конструктора – константа трафарета, значения которой представлены в табл. 6.2.

Таблица 6.2

Стили кисти

Стиль	Описание
HS_HORIZONTAL	Горизонтальная заливка
HS_VERTICAL	Вертикальная заливка
HS_CROSS	Прямая клетка
HS_FDIAGONAL	Диагональная, наклон влево
HS_BDIAGONAL	Диагональная, наклон вправо
HS_DIAGONALCROSS	Косая клетка

В теле цикла в контекст выбирается один из этих трафаретов, определяется положение очередного прямоугольника в поле окна и затем вызывается функция Rectangle(), которая и использует контекст с включенными в него пером и кистью. Затем в контексте восстанавливается прежняя кисть. После выхода из цикла в контексте восстанавливается и перо.

Метод Rectangle() – это один из методов, используемых для построения на экране замкнутых фигур. Rectangle() использует в качестве аргументов

координаты левого верхнего и правого нижнего углов вычерчиваемого прямоугольника. В классе контекста устройства существуют и другие методы: Chord (хорда), Ellipse (эллипс), Pie (сектор), Polygon (многоугольник), PolyPolygon (массив многоугольников), Polyline (линия, соединяющая массив точек), RoundRect (прямоугольник со скругленными углами).

6.6. Рисование графических примитивов в рабочей области окна

В листинге 6.5 представлены примеры рисования простых фигур различными цветами.

Листинг 6.5.

```
void CMainWnd::OnPaint() {
static DWORD dwColor[9] = {RGB(0, 0, 0), // черный
RGB(255,0, 0), // красный
RGB (0,255, 0), // зеленый
RGB(0,0, 255), // синий
RGB(255,255, 0), // желтый
RGB(255,0, 255), // пурпурный
RGB(0,255,255), // голубой
RGB(127,127, 127), // серый
RGB(255,255, 255)};// белый
}
short xcoord;
POINT polylpts[4],polygpts[5] ;
CBrush newbrush;
CBrush* oldbrush;
CPen newpen;
CPen*oldpen;
CPaintDC dc(this);

// рисование эллипса и заливка его красным цветом
newpen.CreatePen(PS_SOLID,
1, dwColor[1]);
oldpen = dc.Selectobject(&newpen);
newbrush.CreateSolidBrush
(dwColor[1]);
oldbrush = dc.Selectobject(&newbrush) ;
dc.Ellipse(180, 180, 285, 260);
dc.TextOut(210,215, "ellipse",7);
// удаление кисти
dc.SelectObject(oldbrush);
newbrush.DeleteObject();
// удаление пера
dc.SelectObject(oldpen);
newpen.DeleteObject{);

// рисование круга и заливка его синим цветом
newpen.CreatePen(PS_SOLID, 1, dwColor[3]);
```

```
oldpen = dc.SelectObject(&newpen);
newbrush.CreatesolidBrush(dwColor[3]) ;
oldbrush = dc.SelectObject(&newbrush);
dc.Ellipse(380,180, 570, 370);
dc.TextOut(450,265,"circle",6);
// удаление кисти
dc.SelectObject(oldbrush) ;
newbrush.DeleteObject();
// удаление пера
dc.SelectObject(oldpen);
newpen.DeleteObject() ;

// рисование черного сектора и заливка его зеленым цветом
newpen.CreatePen(PS_SOLID, 1, dwColor[0]);
oldpen = dc.SelectObject(&newpen);
newbrush.CreatesolidBrush(dwColor[2]);
oldbrush = dc.SelectObject(&newbrush);
dc.Pie(300,50,400, 150, 300, 50,300, 100);
dc.TextOut(350,80,"<-pie wedge", 11);
// удаление кисти
dc.SelectObject(oldbrush);
newbrush.DeleteObject();
// удаление пера
dc.SelectObject(oldpen);
newpen.DeleteObject() ;

// рисование черного прямоугольника и заливка его серым цветом
newbrush.CreateSolidBrush(dwColor[7]) ;
oldbrush = dc.SelectObject(&newbrush);
dc.Rectangle(50,300, 150, 400);
dc.TextOut(160,350,"<-rectangle", 11);
// удаление кисти
dc.SelectObject(oldbrush);
newbrush.DeleteObject() ;

// рисование голубого многоугольника
//и заливка его диагональной желтой штриховкой
newpen.CreatePen(PS_SOLID, 4, dwColor[6]);
oldpen = dc.SelectObject(&newpen);
newbrush.CreateHatchBrush(HS_FDIAGONAL, dwColor[4]);
oldbrush = dc.SelectObject(&newbrush);
polygpts[0].x = 40;
polygpts[0].y = 200;
polygptsfl].x = 100;
polygpts[l].y = 270;
polygpts[2].x = 80;
polygpts[2].y = 290;
polygpts[3].x = 20;
polygpts[3].y = 220;
polygpts[4].x = 40;
polygpts[4].y = 200;
dc.Polygon(polygpts, 5);
```

```
dc.TextOut(70,210,"<-polygon", 9) ;
// удаление кисти
dc.SelectObject(oldbrush);
newbrush.DeleteObject();
// удаление пера
dc.SelectObject(oldpen);
newpen.DeleteObject();
}
```

Для задания цвета пера или кисти удобно пользоваться стандартной панелью выбора цвета:

```
CColorDialog cc; //создать объект класса стандартной диал. пане-
ли
if (cc.DoModal()==IDOK) //отобразить панель на экране
m_color=cc.GetColor();//принять выбранный цвет в переменную
                        //m_color
```

7. Сохранение и восстановление состояния объектов

Одна из основных задач программы – сохранять данные пользователя после их изменения по той или иной причине. Без этого вся работа, которую пользователь затратил на редактирование данных, пропадет, как только приложение завершит работу. Когда приложение создается с использованием AppWizard, Visual C++ включает в него программы, которые необходимые для сохранения и восстановления данных.

Одна из задач, решаемых программистом при разработке приложений, которые могут создавать и редактировать документы различных типов, заключается в том, чтобы предоставить пользователю возможность записать внутреннее представление документа в файл и восстановить его. Процесс сохранения и восстановления внутреннего представления объекта называется сериализацией данных.

При создании приложения приходится иметь дело с достаточно большим разнообразием типов объектов. Одни типы объектов, хранящих данные, довольно просты, например тип int или char, другие являются экземплярами классов. При использовании таких объектов в приложении, которое должно формировать, сохранять и восстанавливать документы, разработчику необходимо разрабатывать средства сохранения и восстановления этих объектов, с тем чтобы можно было их восстановить. Свойство объекта сохраняться и восстанавливаться называется живучестью (persistence). Практически все классы MFC наделены этим свойством, поскольку они прямо или косвенно происходят от базового класса CObject. Последний уже обладает базовыми функциями сохранения-восстановления объекта.

Приложения, подготовленные при помощи средства AppWizard, используют этот механизм с помощью методов класса CDocument. Программисту предлагается только переопределить метод Serialize этого класса для работы с конкретными данными приложения. Кроме этого, программист может определить свой класс (на основе базового класса CObject) для работы с данными и определить в нем механизм записи и восстановления объектов.

7.1. Создание класса, обеспечивающего сериализацию данных

Библиотека классов MFC определяет механизм записи и восстановления объектов (serialization), причем поддержка этого механизма осуществляется средствами класса CObject. Классы, наследованные от CObject, могут обеспечивать работу механизма записи и восстановления объектов. Для этого при объявлении класса надо указать макрокоманду DECLARE_SERIAL, а при определении – макрокоманду IMPLEMENT_SERIAL.

Макрокоманду DECLARE_SERIAL необходимо поместить в описании класса в заголовочном файле. В качестве параметра макрокоманды надо указать имя класса:

```
DECLARE_SERIAL (имя_класса)
```

Макрокоманду IMPLEMENT_SERIAL следует указать перед упоминанием класса в файле исходного текста приложения. Прототип макрокоманды IMPLEMENT_SERIAL представлен ниже:

```
IMPLEMENT_SERIAL (имя_класса, имя_базового_класса, номер_версии)
```

Параметр «имя_базового_класса» определяет имя базового класса, от которого непосредственно наследуется класс. Параметр «номер_версии» – это число типа UINT, определяющее версию программы.

В классе должны быть определены специальные методы для записи и восстановления состояния объектов этого класса. Обычно эти методы сохраняют и восстанавливают элементы данных из класса. Таким образом, объекты класса сами отвечают за то, как они сохраняют и восстанавливают свое состояние. Методы, сохраняющие и восстанавливающие объекты, взаимодействуют с объектом класса CArchive, который осуществляет непосредственную запись и чтение информации из файла на диске.

Класс CObject содержит виртуальный метод Serialize, отвечающий за запись и чтение объектов классов, наследованных от класса CObject:

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

В качестве параметра ar методу передается указатель на объект класса CArchive, используемый для записи и восстановления состояния объекта класса CObject (или наследуемого от него класса). Чтобы узнать, какую операцию должен выполнить метод Serialize, необходимо воспользоваться методами IsLoading или IsStoring класса CArchive.

Итак, при создании нового класса, в котором метод Serialize применяется для сериализации данных, необходимо:

– чтобы класс был производным от класса CObject или его потомков;

– при объявлении класса вставить макрокоманду DECLARE_SERIAL;

– определить в классе функцию Serialize, отвечающую за хранение переменных класса;

– определить в классе конструктор без параметров. Это может быть защищенный конструктор, если он вызывается только для процесса сериализации данных. В конструкторе возможно динамическое создание объектов и инициализация переменных, если это необходимо;

– объявить в классе деструктор, если требуется выполнить специальные действия при разрушении объектов класса, например освободить память динамически созданных объектов;

– в начало файла реализации класса вставить макрос IMPLEMENT_SERIAL.

7.2. Сериализация в классе документа

Рассмотрим SDI-приложение My, построенное AppWizard и поддерживающее архитектуру «документ – представление». Определим данные, с которыми работает представление. Добавим в секцию атрибутов класса CMyDoc (в файле MyDoc.h) определение переменной m_message, типа CString, чтобы этот фрагмент определения класса выглядел следующим образом:

```
public:
CString m_message;
```

В данном случае документ содержит единственный объект класса CString – строку текста. В реальных приложениях данные будут значительно сложнее. Однако этой единственной строки текста хватит, чтобы продемонстрировать особенности технологии обеспечения сохранности документа. Очень часто программисты используют в классе документа открытые члены-переменные вместо закрытых членов, для каждого из которых организуется открытая функция доступа. Это несколько облегчает разработку класса представления, методы которого должны обращаться к членам класса документа. Но в дальнейшем при сопровождении и модификации программы такой подход несколько усложнит жизнь.

Класс документа должен обеспечить инициализацию данных при открытии нового документа, что возлагается на метод OnNewDocument() этого класса. Вызовите при помощи окна ClassView текст этой функции в окно редактора кода и отредактируйте его. Добавьте оператор инициализации строковой переменной.

```
BOOL CMyDoc::OnNewDocument()
{
if(!CDocument::OnNewDocument())
  return FALSE;
m_message = "Default Message";
  return TRUE;
}
```

После того как переменная m_message – член класса документа – инициализирована, приложение должно вывести содержимое документа в свое окно. За вывод содержимого документа отвечает метод OnDraw() класса представления. После редактирования метод OnDraw() должен иметь вид:

```
void CMyView: :OnDraw(CDC* pDC)
{
CMyDoc* pDoc = GetDocument();
ASSERT_VALID(pDoc);
pDC->TextOut(20,20, pDoc->m_message);
}
```

Если оттранслировать приложение My и запустить его на выполнение, на экране появится сообщение Default Message. Теперь необходимо обеспечить возможность редактирования. Для этого нужно включить в меню Edit приложения пункт Change Message. По этой команде должны запускаться средства, позволяющие пользователю изменить текст документа – выводимого сообщения.

Создайте обработчик события выбора пункта меню Change Message:

```
Void CMyView::OnEditChangemessage()
{
CTime now = CTime::GetCurrentTime();
CString changetime = now.Format("Changed at %B %d %H:%M:%S");
GetDocument()->m_message = changetime;
GetDocument()->SetModifiedFlag();
Invalidate();
}
```

Эта функция формирует строку соответственно текущей дате и времени и присваивает ее переменной-члену текущего объекта класса документа. Вызов метода SetModifiedFlag() класса документа сообщит приложению, что содержимое документа изменено. Если такое изменение зафиксировано, приложение будет предупреждать пользователя о наличии несохраненных изменений в текущем документе при попытке его закрыть. И последняя операция – запуск механизма обновления представления документа на экране, который производится функцией Invalidate().

Если m_message является закрытой переменной-членом класса документа, понадобится разработать открытый метод SetMessage(), который будет самостоятельно вызывать SetModifiedFlag(). Таким образом, можно не напоминать программистам об обязательном обращении к этой функции при модификации объекта класса документа. В этом и состоит преимущество следования принципам объектно-ориентированного программирования.

Метод Serialize() класса документа должен позаботиться о сохранении-восстановлении данных документа. Текст заготовки функции Serialize(), сформированный AppWizard, выглядит следующим образом:

```
void CMyDoc::Serialize(CArchive& ar)
{
if (ar.IsStoring())
  {
// TODO: сюда вставьте операторы сохранения данных.
  }
else
  {
//TODO: сюда вставьте операторы загрузки данных.
  }
}
```

Поскольку в классе CString (объектом которого является переменная m_message) определены терминальные операторы >> и << для передачи данных в архив и из него, это значительно упрощает сохранение и восстановление данных в объекте класса документа. Добавьте следующий оператор в том месте, где в заготовке стоит инструктирующий комментарий:

```
ar<< m_message;
```

Аналогично в том месте текста программы, где должны стоять операторы загрузки, вставьте:

```
ar>> m_message;
```

Указанные терминальные операторы определены для следующих простых типов данных: BYTE, WORD, int, LONG, DWORD, float, double.

Выбор в меню пункта Edit→ChangeMessage вызовет появление на экране новой строки сообщения. Сохраните документ с помощью File→Save. Измените текст сообщения с помощью команды Edit→ChangeMessage. Выберите File→New, на экране появится предупреждающее сообщение о наличии в документе несохраненных изменений. Вам будет предложено сохранить их на диске прежде, чем открывать новый документ. Теперь выберите File→0pen и введите имя ранее созданного файла документа (его можно найти и в списке в самом низу меню File). После этого на экране появится ранее сохраненный текст сообщения. Таким образом, можно убедиться, что приложение My сохранило документ, а затем восстановило его в прежнем виде.

8. Диалоги. Классы окон. Элементы управления

8.1. ClassWizard и диалоговые окна

В операционной среде Windows получить данные от пользователя приложение может через диалоговые окна. Приложение может иметь любое количество диалоговых окон, в которых происходит ввод данных пользователем. Библиотека классов MFC содержит класс CDialog (наследованный от базового класса CWnd), специально предназначенный для управления диалоговыми панелями.

Диалоговые панели бывают двух типов – модальные и немодальные. Большинство диалоговых окон, которые приходится включать в состав приложения, относятся к модальным окнам. Модальное окно выведено всегда поверх всех остальных окон на экране. Пользователь должен поработать в этом окне и обязательно закрыть его, прежде чем приступить к работе в любом другом окне этого же приложения. Примером может служить окно, которое открывается при выборе команды File→Open любого приложения Windows. Немодальное диалоговое окно позволяет пользователю, не закончив работы с ним, работать в других окнах приложения, выполнить там необходимые действия, затем снова вернуться в немодальное окно и продолжить работу. Типичными немодальными окнами являются окна, которые открываются при отработке команд Edit→Find и Edit→Replace во многих приложениях Windows.

Как правило, для каждого диалогового окна в приложении необходимо разработать две вещи: ресурсы окна и класс окна. Ресурсы окна используются программой для того, чтобы вывести на экран его изображение и изображения элементов управления, которые входят в него. В класс окна включены переменные и функции-члены, ответственные за работу диалога. Каждая диалоговая панель обычно содержит несколько органов управления. Работая с диалоговой панелью, пользователь взаимодействует с этими органами управления – нажимает кнопки, вводит текст, выбирает элементы списков. В результате генерируются соответствующие сообщения, которые должны быть обработаны классом диалоговой панели. Так как класс диалоговой панели обрабатывает сообщения, то содержит таблицу сообщений и соответствующие методы-обработчики сообщений.

Ресурсы диалога создаются посредством редактора ресурсов, с помощью которого возможно включать в его состав необходимые элементы управления и размещать их в необходимом порядке. Класс окна создается при помощи ClassWizard. Как правило, класс диалогового окна в проекте является производным от класса CDialog, входящего в MFC. ClassWizard также поможет связать ресурсы окна с классом окна.

Обычно каждый элемент управления, включенный в состав ресурсов диалога, имеет в классе окна соответствующий член-переменную. Для того

чтобы вывести диалоговое окно на экран, нужно вызвать функцию – член его класса. Для того чтобы установить значения по умолчанию для элементов управления перед выводом окна на экран или считать состояние элементов управления после завершения работы пользователя, необходимо обращаться к членам-переменным класса.

Чтобы создать модальную диалоговую панель, сначала необходимо создать объект определенного в приложении класса диалоговой панели, а затем вызвать метод DoModal, определенный в классе CDialog.

8.2. Формирование нового ресурса диалогового окна

Первый шаг процесса организации диалогового окна в приложении – формирование ресурса окна. Создадим SDI-приложение с именем lab1. Чтобы приступить к формированию ресурсов, необходимо выбрать пункт Add Resource в контекстном меню файла lab1.rc. Появится диалоговое окно Add Resource (рис. 8.1).

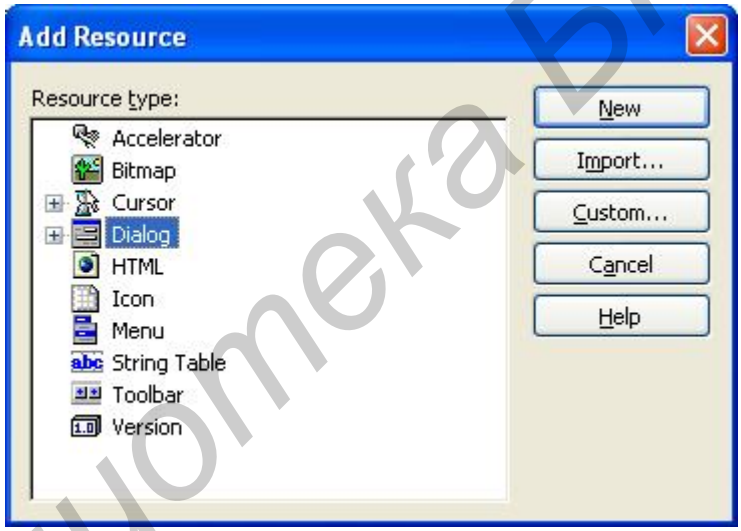

Рис. 8.1. Добавление ресурса диалогового окна

Дважды щелкните на элементе Dialog – этим вы вызываете редактор диалогового окна, который выводит на экран заготовку окна, как это показано на рис. 8.2. Вызовите на экран диалоговое окно Dialog Properties для вновь создаваемого диалогового окна, щёлкнув правой кнопкой мыши по диалоговому окну и выбрав Properties. В поле Caption (Надпись) введите заголовок (например LAB1) диалога (см. рис. 8.2).

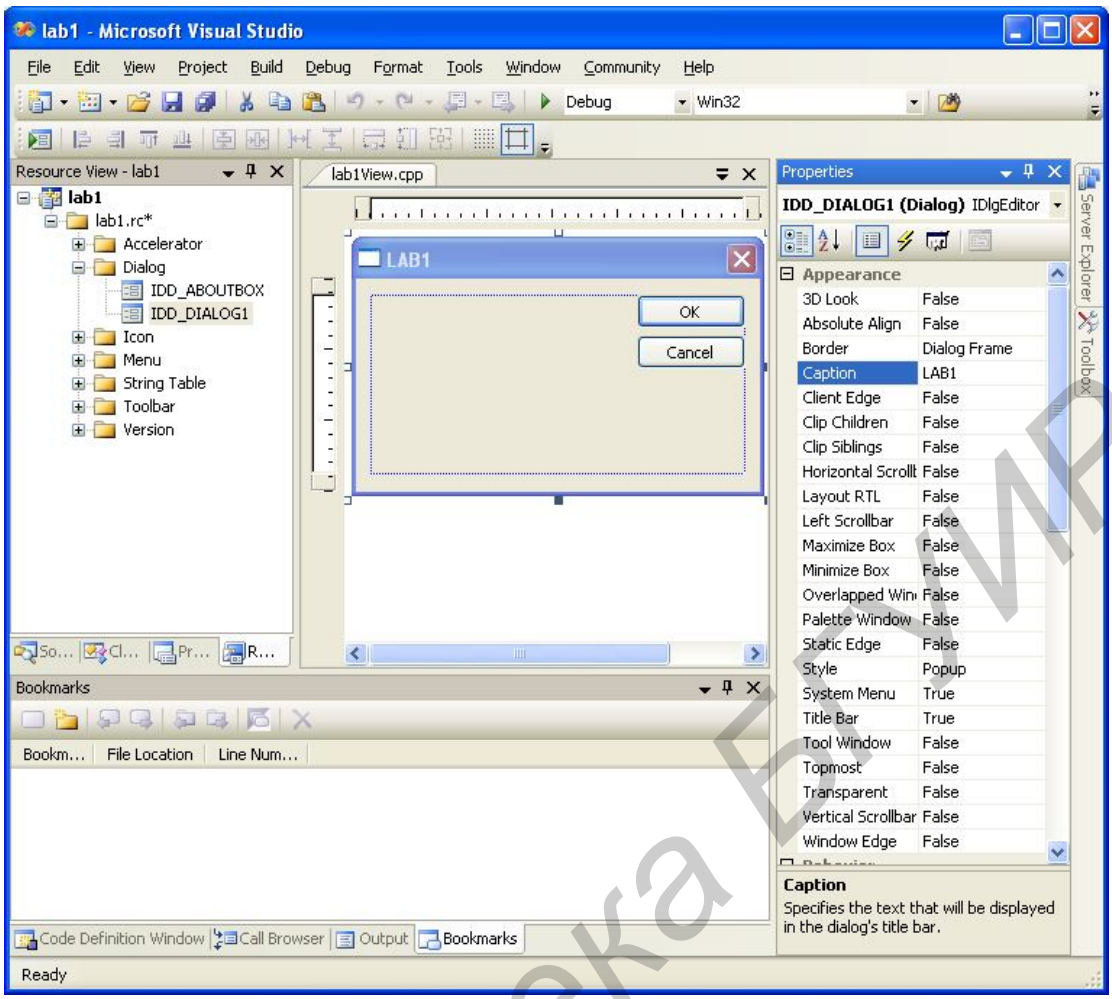

Рис. 8.2. Заготовка диалогового окна

8.3. Создание класса диалогового окна

Когда формирование ресурсов диалогового окна будет завершено, сделайте по нему двойной клик и вызовите на экран диалоговое окно мастера ClassWizard. Мастер ClassWizard обнаружит новый диалог и предложит создать новый класс. Появится диалоговое окно, которое показано на рис. 8.3. В поле Class name введите имя нового класса (например CLab1Dlg) и щелкните на Finish. После этого ClassWizard создаст новый класс, подготовит файл текста программы Lab1Dlg.cpp и файл заголовка Lab1Dlg.h и включит их в состав проекта.

Выбор элементов управления для диалогового окна достаточно велик. Для установки элементов диалогового окна используется технология, получившая наименование WYSIWYG (What You See Is What You Get): что видишь, то и получишь. Чтобы установить в своем окне кнопку, нужно выбрать ее образец на поле инструментария, перетянуть в желаемое место на поле заготовки окна приложения, сбросить ее там и заменить надпись.

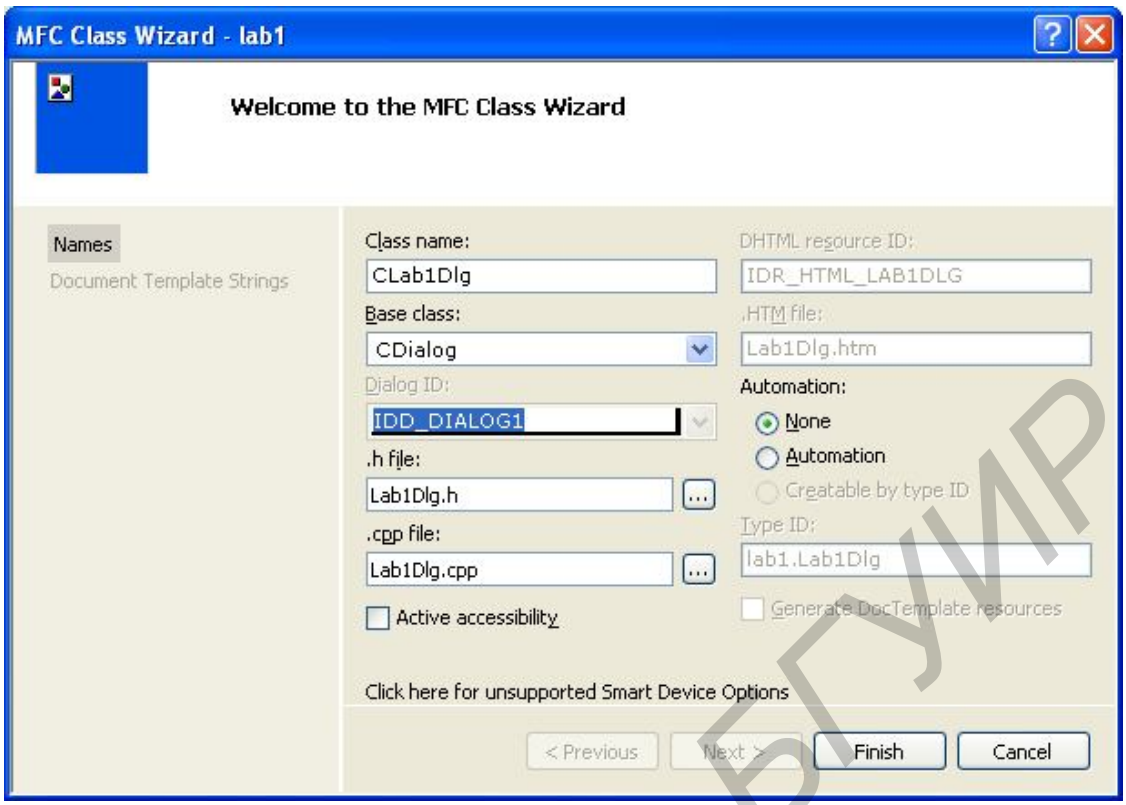

Рис. 8.3. Создание класса для диалогового окна

Краткая характеристика основных Windows-элементов управления, используемых для построения диалога:

– надпись (static text) – по существу, это «неполноценный» элемент управления, поскольку он используется только как поле для вывода надписи, относящейся к «настоящему» элементу управления, расположенному рядом;

– текстовое поле (edit box) – может быть однострочным или многострочным; сюда пользователь может ввести текст;

– кнопка (button) – данный элемент предназначен для начала каких-либо действий;

– флажок (check box) – используется для установки опций, каждая из которых может быть выбрана независимо от других;

– переключатель-радиокнопка (radio button) – используется для выбора одной из групп связанных опций; если выбрана одна из них, то другие полагаются невыбранными;

– список (list box) – используется для выбора одного элемента из заранее подготовленного набора; набор может быть как жестко установленным на этапе разработки программы, так и меняться программно в процессе выполнения приложения; главное – пользователь по своей воле не может непосредственно менять элементы в наборе, он может только их выбирать;

– поле со списком (combo box) – это комбинация текстового поля и списка; такой элемент управления позволяет пользователю не только выби-

рать элементы из ранее подготовленного набора, но и самостоятельно пополнять его, непосредственно внося необходимый текст в текстовое поле.

Для примера установим элементы управления на поверхности диалога, как показано на рис. 8.4.

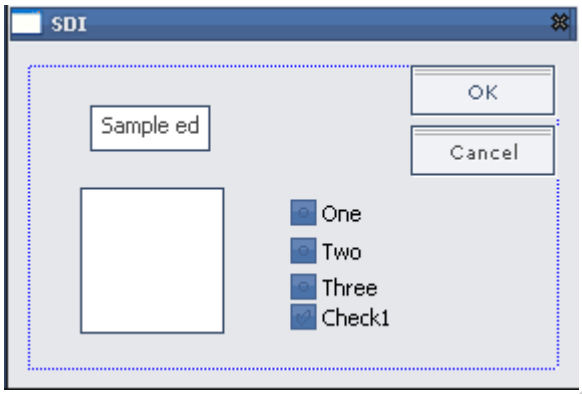

Рис. 8.4. Диалоговое окно с элементами управления

8.4. Задание идентификаторов диалогового окна и элементов управления

Поскольку каждое диалоговое окно в приложении является уникальным объектом, разработчику практически всегда нужно присваивать окнам и элементам управления, входящим в их состав, идентификаторы по собственному выбору. Конечно, можно согласиться и с теми идентификаторами, которые предлагает редактор диалоговых окон по умолчанию. Они не несут смысловой нагрузки (как правило, нечто вроде IDD_DIALOG1, IDC_EDIT1, IDC_RADIO1) и их можно заменить другими, связанными с назначением и функциями окна или элемента. Но в любом случае рекомендуется соблюдать соглашение о префиксах: идентификаторы диалоговых окон имеют префикс IDD_, а идентификаторы элементов управления – IDC_. Заменить идентификатор можно с помощью поля ID диалогового окна Properties.

8.5. Создание ассоциированных переменных

Ассоциированная переменная позволяет установить связь между текстом программы и ресурсами окна. Она задается следующим образом: выбираем элемент управления, нажимаем правой кнопкой мыши и вызываем окно Add Member Variables (рис. 8.5). Пример, представленный на рис. 8.6, демонстрирует один из вариантов ассоциативной связи. Элементу IDC_CHECK1 следует присвоить идентификатор переменной m_check. Нужно проверить, чтобы в раскрывающемся списке Category было выбрано Value. Если вы раскроете список Variable type, то увидите, что вам предоставлен единственный выбор – BOOL. Флажок может быть либо установлен, либо сброшен, а значит, ассоциирован только с переменной типа BOOL, которая принимает

только два значения – TRUE и FALSE. Нажмите на OK для завершения процедуры.

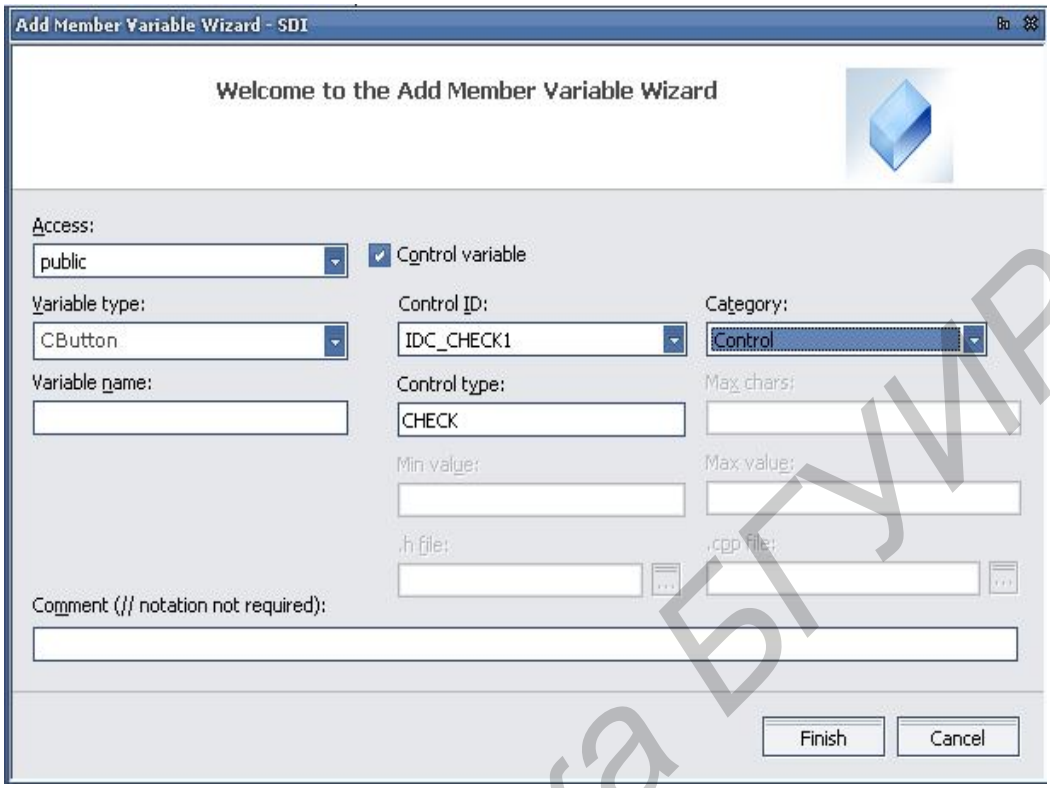

Рис. 8.5. Окно Add Member Variables

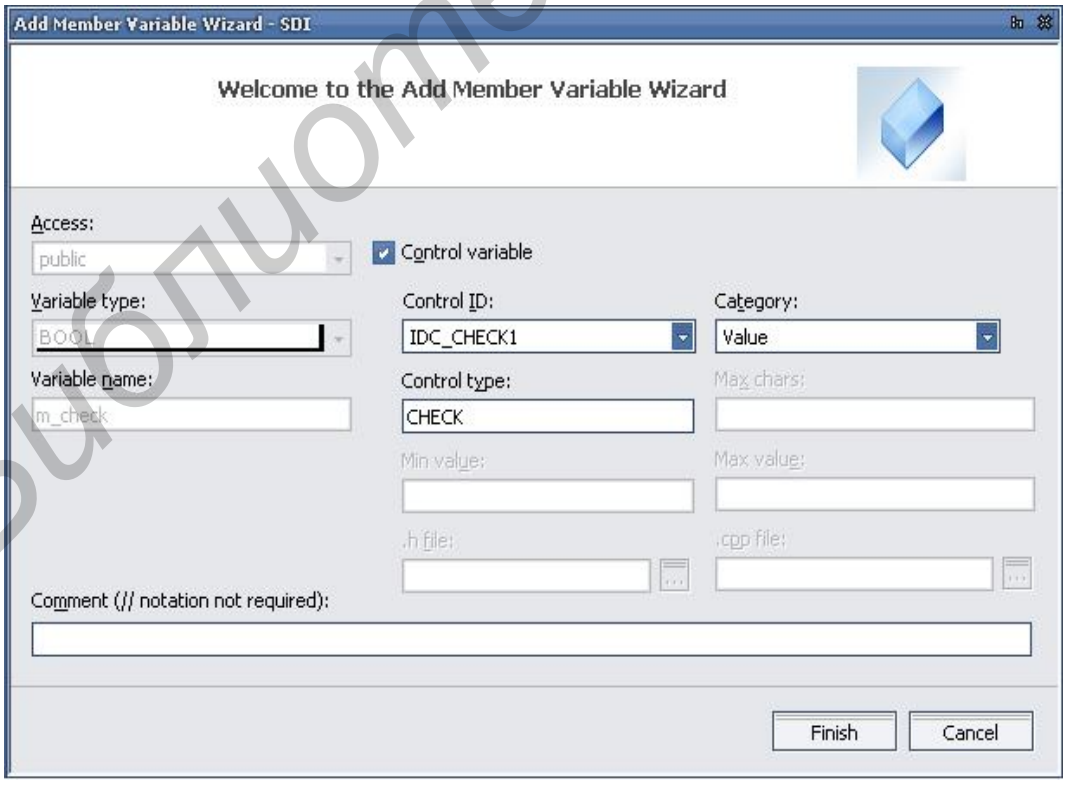

Рис. 8.6. Диалог установки имени ассоциированной переменной

Свяжите таким же образом значение, которое содержится в элементе IDC_EDIT1, с членом-переменной m_edit типа CString. Элемент IDC_LIST1 должен быть связан с членом-переменной m_list, который должен быть объектом класса CListBox (в списке Category должно быть избрано Control). Первый переключатель в группе IDC_RADIO1 должен быть связан с членом-переменной m_radio типа int, причем связь должна быть установлена по значению.

8.6. Организация вывода диалогового окна на экран

Теперь, когда сформирован ресурс и подготовлен класс окна, можно создавать объект этого класса в программе и выводить на экран связанное с ним диалоговое окно. Выведем диалоговое окно сразу после запуска приложения (в методе InitInstance главного класса приложения). Для этого используется функция DoModal().

Предположим, мы создали проект типа «Single document» под названием lab1 и хотим вызвать наш диалог в главном классе приложения CLab1App. Выполним это в функции InitInstance(). Эта функция вызывается при любом запуске приложения. Наш диалог называется CLab1Dialog и расположен в файлах Lab1Dialog.h и Lab1Dialog.cpp. Перейдем в самое начало файла и после уже имеющихся директив # include вставим еще одну:

```
#include "Lab1dialog.h"
```

Теперь при трансляции компилятор будет знать, где взять информацию о классе CLab1Dialog. Перейдем в конец текста функции CSDIApp::InitInstance() в файле SDI.cpp и добавим перед окончанием текста функции следующие строки, представленные в листинге 8.1.

Листинг 8.1. Фрагмент функции InitInstance().

```
CLab1Dialog dlg;
dlg.m_check = TRUE;
dlg.m_edit = "hi there";
CString msg;
if (dlg.DoModal() == IDOK) {
msg = "You clicked OK. ";}
else
{msg = "You clicked Cancel. ";}
msg += "Edit Box is: ";
msg += dlg.m_edit;
AfxMessageBox(msg);
```

Приведенный выше фрагмент создает экземпляр класса диалогового окна. Он устанавливает параметры по умолчанию для двух элементов управления – флажка и текстового поля. Сам по себе вывод диалогового окна про-

изводится функцией DoModal(), которая возвращает числовое значение – IDOK, если пользователь вышел из окна, нажав на OK, и IDCANCEL, если выход произошел после нажатия на Cancel. Затем в приведенном фрагменте формируется сообщение, которое выводится на экран функцией AfxMessageBox().

Когда вы щелкнули на OK, MFC организовал вызов функции OnOK(). Функция OnOK() унаследована от базового класса CDialog, классом-наследником которого является наш диалог. Помимо прочего в нем находится функция DoDataExchange(), подготовленная средствами ClassWizard (листинг 8.2).

Листинг 8.2. Функция DoDataExchange().

```
void CLab1Dialog::DoDataExchange(CDataExchange* pDX)
{CDialog::DoDataExchange(pDX);
//{{AFX_DATA_MAP(CSdiDiaIog)
DDX_Control(pDX, IDC_LIST1, m_list);
DDX_Check(pDX, IDC_CHECK1, m_check);
DDX_Text(pDX, IDC_EDIT1, m_edit);
DDV_MaxChars(pDX, m_edit, 10);
DDX_Radio(pDX, IDC_RADI01, m_radio);
//}}AFX_DATA_MAP
}
```

Все функции, имена которых начинаются с DDX, выполняют обмен данными. Первым аргументом является указатель на объект класса CDataExchange, этот объект определяет параметры обмена, в том числе направление, в котором надо выполнить обмен данными. Вторым аргументом каждой функции является идентификатор элемента управления, а третьим – переменная класса. Именно таким образом ClassWizard устанавливает соответствие между элементами управления и членами класса диалогового окна. ClassWizard также включил имена членов-переменных в файл заголовка, в котором объявляется структура класса.

Имеются 34 функции, их имена начинаются с DDX – одна на каждый тип данных, которыми могут обмениваться диалоговое окно и соответствующий класс. Каждая функция включает в свое имя также имя элемента управления. Например, функция DDX_Check() используется для связи между элементом типа флажок (check box) и членом-переменной типа BOOL. Аналогично DDX_Text() используется для связи члена-переменной типа CString с текстовым полем. ClassWizard выбирает соответствующую функцию в процессе выполнения описанной выше операции связывания членов класса с элементами управления.

Существует несколько DDX-функций, за которые ClassWizard не несет ответственности. Например, если вы связываете список по значению с переменной, то единственным выбором для вас является тип CString. В этом случае ClassWizard формирует функцию DDX_LBString(), которая связывает

выбранный в списке элемент с членом-переменной типа CString. Однако иногда эффективнее использовать индекс выбранного элемента, а не сам элемент. Для этого случая имеется функция DDX_LBIndex(), которая выполняет соответствующий обмен. Вызов этой функции можно добавить в текст функции-члена DoDataExchange. Соответствующая строка может быть вставлена в том месте программы, где имеется специальный комментарий, созданный ClassWizard. При этом не забудьте добавить соответствующую переменную-член в объявление класса в файле заголовка. Полный список DDX-функций можно найти в электронной документации.

Функции, имена которых начинаются с DDV, ответственны за проверку соблюдения заданных ограничений на вводимые данные. ClassWizard вставляет вызов DDV_MaxChars() сразу за вызовом DDX_Text, которая передает содержимое текстового поля IDC_EDIT1 в переменную m_edit. Второй аргумент вызова функции – идентификатор переменной-члена, а третий – значение параметра, ограничивающего длину вводимой строки. Если пользователь когда-нибудь при работе с программой попытается ввести символов больше, чем дозволено, то DDV_MaxChars() организует вывод на экран предупреждающего сообщения.

Приложение не должно напрямую вызывать метод DoDataExhange. Он вызывается через метод UpdateData, определенный в классе CWnd. Необязательный параметр этой функции определяет, как будет происходить обмен данными. Если метод UpdateData вызывается с параметром FALSE, то выполняется инициализация диалоговой панели. Информация из данных класса отображается в органах управления диалоговой панели.

В случае если метод UpdateData вызван с параметром TRUE, данные перемещаются в обратном направлении. Из органов управления диалоговой панели они копируются в соответствующие элементы данных класса диалоговой панели.

Метод UpdateData возвращает ненулевое значение, если обмен данными прошел успешно, и нуль в противном случае. Ошибка при обмене данными может произойти, если данные копируются из диалоговой панели в элементы класса диалоговой панели, и пользователь ввел неправильные данные, отвергнутые процедурой автоматической проверки данных.

При создании модальной диалоговой панели перед тем, как панель появится на экране, вызывается виртуальный метод OnInitDialog класса CDialog. По умолчанию OnInitDialog вызывает метод UpdateData и выполняет инициализацию органов управления. Если метод OnInitDialog переопределяется в классе диалоговой панели, в первую очередь необходимо вызвать метод OnInitDialog класса CDialog. Метод UpdateData также вызывается некоторыми другими методами класса CDialog. Так, метод UpdateData вызывается, когда пользователь закрывает модальную диалоговую панель, нажимает кнопку OK. Заметим, что кнопка OK должна иметь идентификатор IDOK. Если пользователь нажмет на кнопку Cancel, имеющую идентификатор

IDCANCEL, то диалоговая панель также закрывается, но метод UpdateData не вызывается и обмен данными не происходит.

Методу DoDataExchange, который служит для реализации механизмов автоматического обмена данными, передается указатель pDX на объект класса CDataExchange. Этот объект создается, когда инициируется процесс обмена данными вызовом функции UpdateData. Элементы данных класса CDataExchange определяют процедуру обмена данными, в том числе определяют, в каком направлении будет происходить этот обмен. Следует обратить внимание на то, что указатель pDX передается функциям DDX_ и DDV_.

8.7. Примеры программирования диалоговых окон

8.7.1. Удаление, добавление, редактирование элементов списка combo box

Рассмотрим операции добавления и редактирования элемента списка типа combo box. Создадим проект SDI типа «Dialog based». Расположим на диалоге элементы управления: список combo box, поле редактирования edit box, кнопку Add с идентификатором IDC_ADD (рис. 8.7).

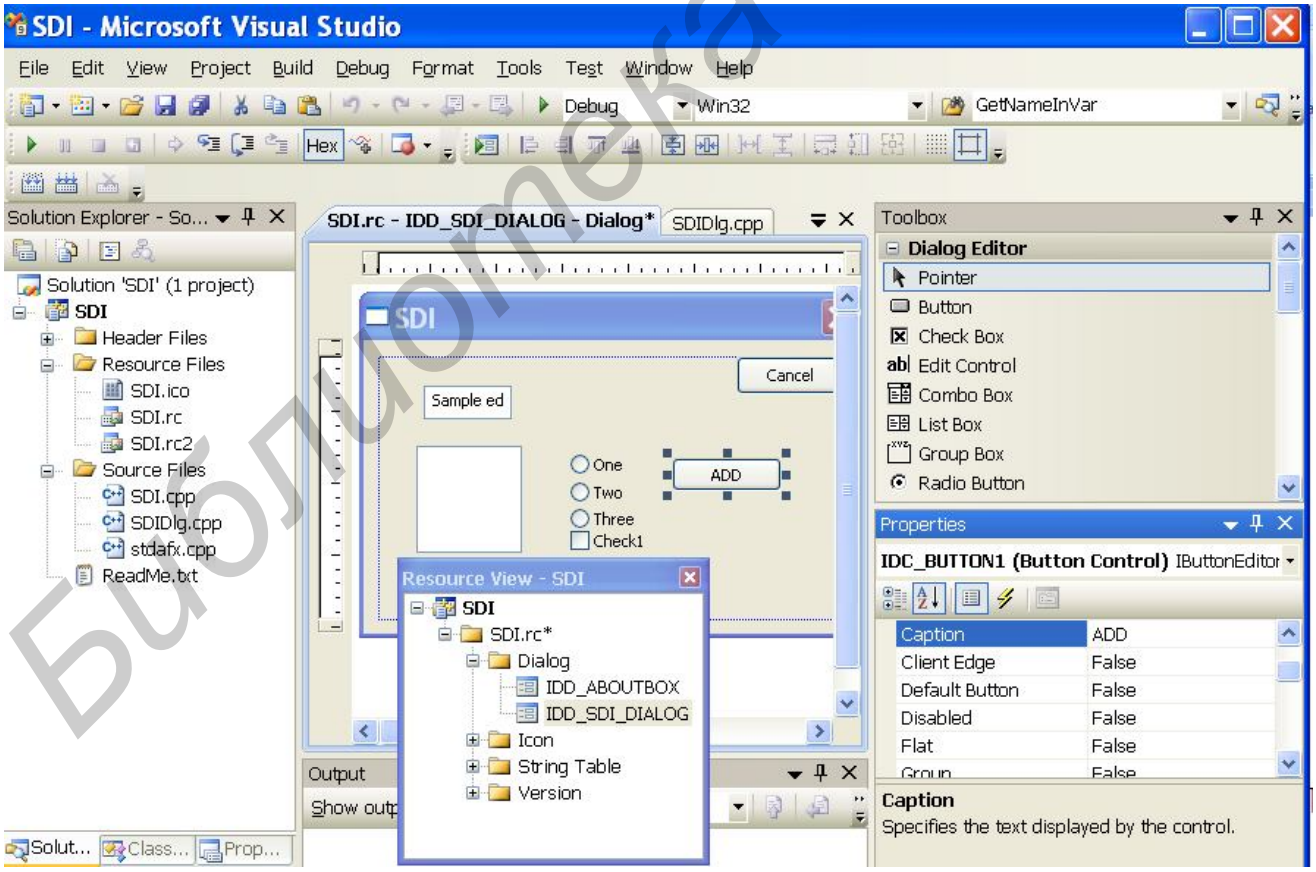

Рис. 8.7. Настройка свойств IDC_ADD

Добавим обработчик сообщения для кнопки ADD BN_CLICKED – по двойному клику на ресурсе «ADD» (рис. 8.8). Настроим свойства элемента Combo box (рис. 8.9 и 8.10). Создадим ассоциированную переменную. Для этого вызовем контекстное меню для элемента «ADD» и выберем в нем пункт «Add variable» (рис. 8.11).

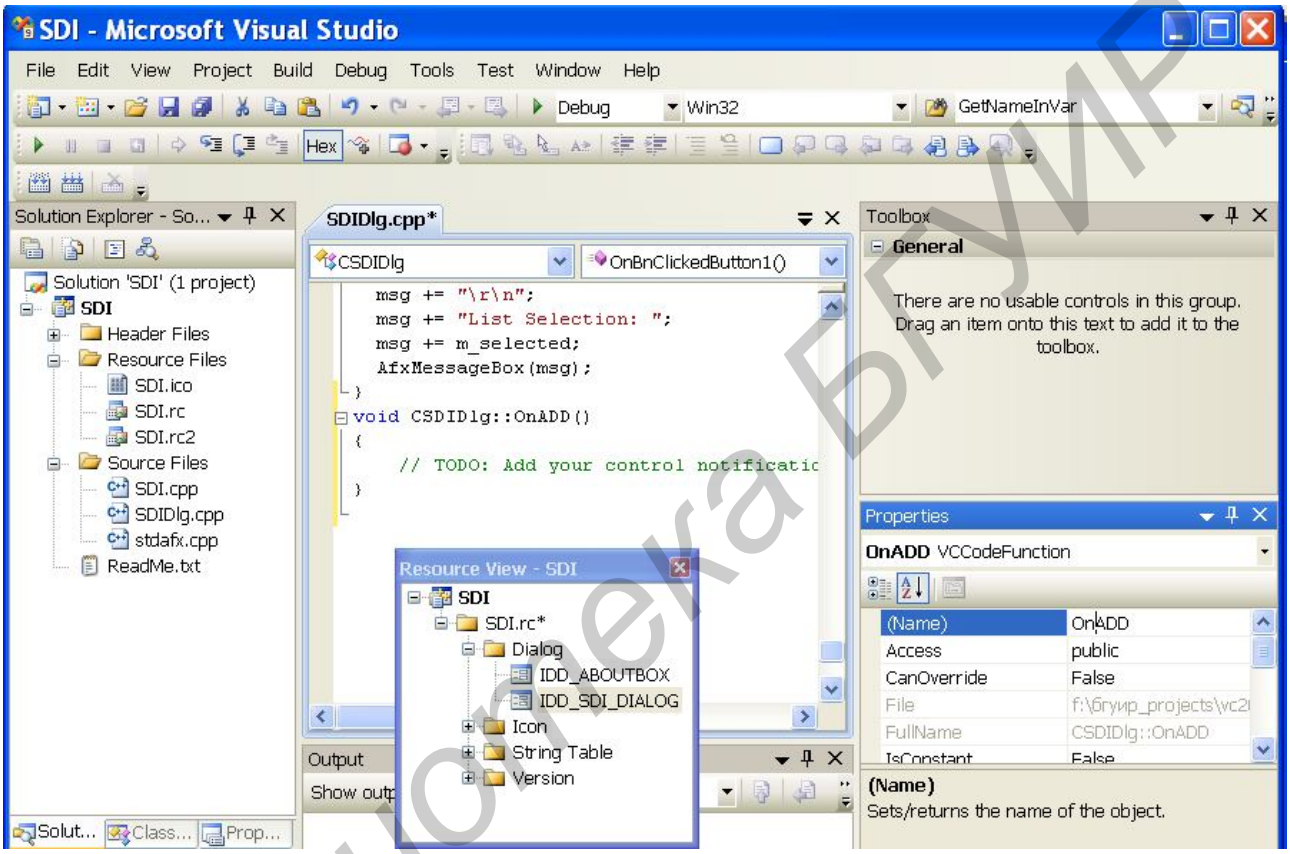

Рис. 8.8. Добавление обработчика события OnADD

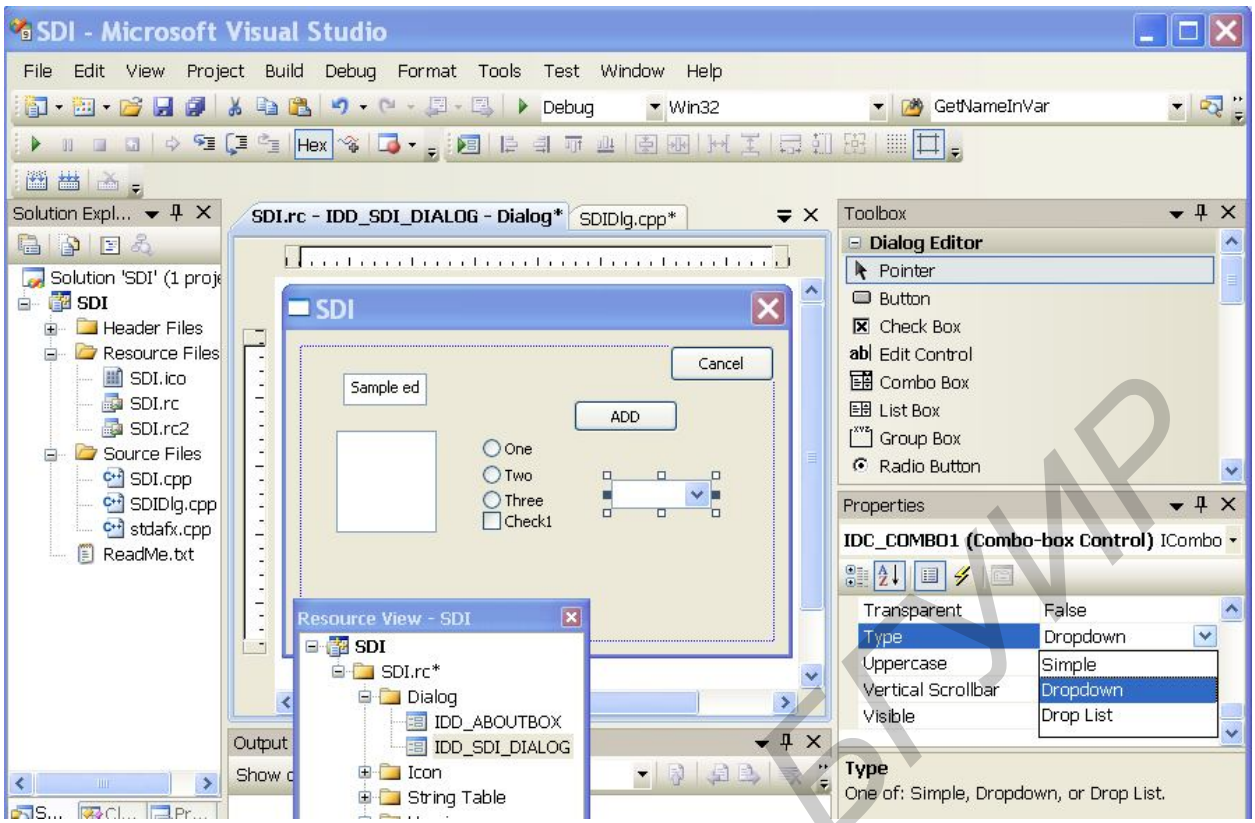

Рис. 8.9. Настройка свойств элемента Combo box

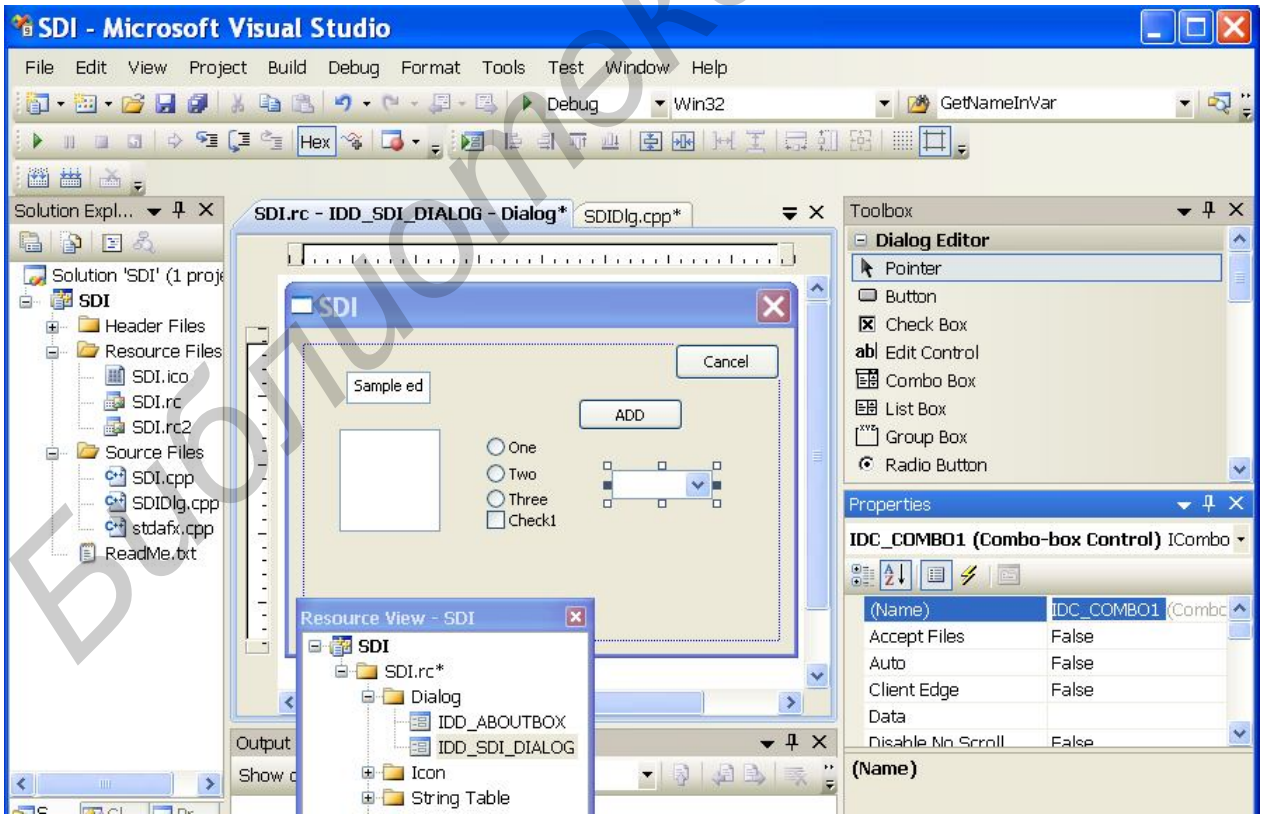

Рис. 8.10. Настройка свойств элемента Combo box

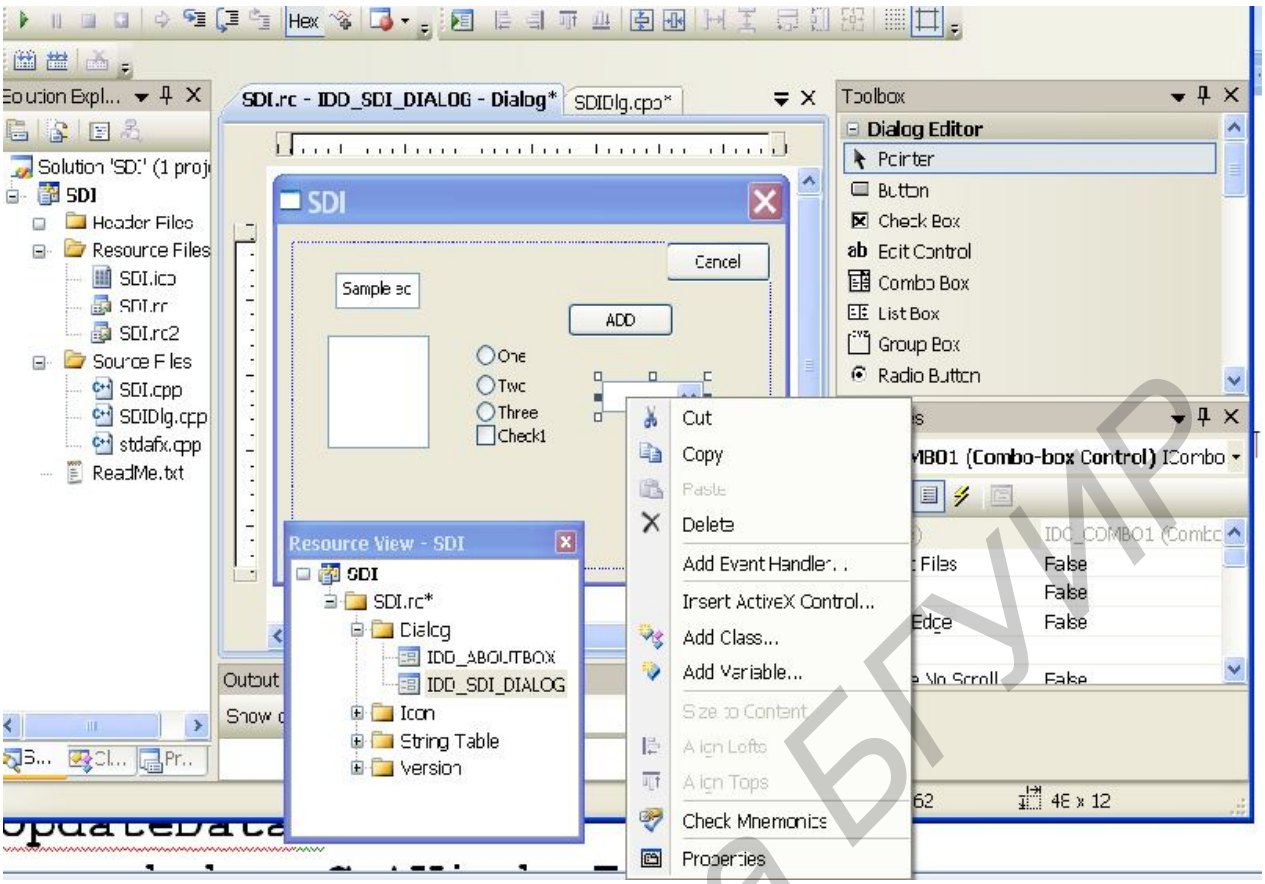

Рис. 8.11. Выбор пункта меню «Add Variable»

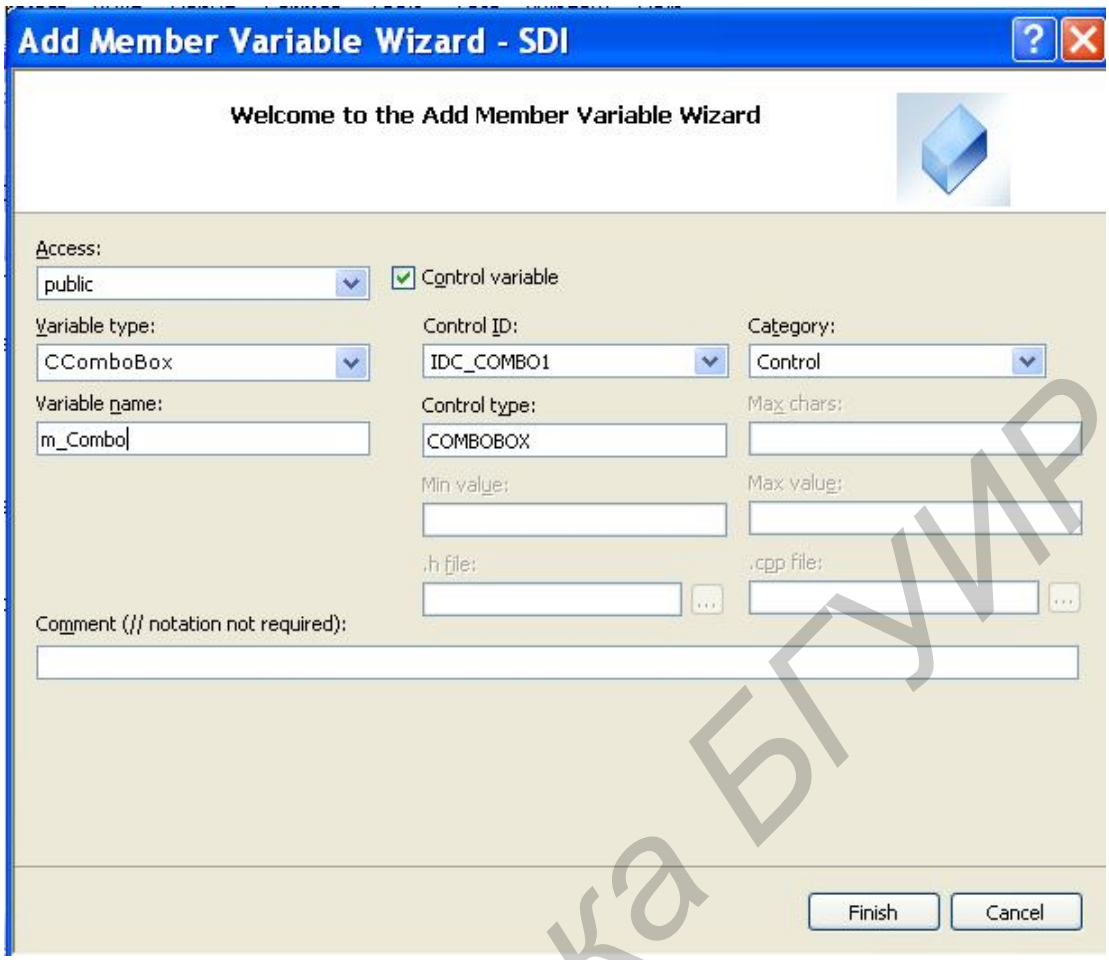

Рис. 8.12. Создание ассоциированной переменной m_combo

Отредактируем функцию OnAdd(), как представлено в листинге 8.3.

Листинг 8.3.

```
void CSDIDlg::OnAdd()
{
CString str;
UpdateData();
m_combo.GetWindowText(str);
m_combo.AddString(str);
UpdateData(FALSE);
}
```

Чтобы выполнить операцию удаления элемента из списка Combo box, необходимо создать кнопку DELETE и запрограммировать по аналогии с ADD (листинг 8.4).

Листинг 8.4.

```
void CSDIDlg::OnDelete()
{
UpdateData();
```

```
m_combo.DeleteString
(m_combo.GetCurSel());
UpdateData(FALSE);
}
```

Чтобы выполнить операцию редактирования элемента из списка Combo box, необходимо создать кнопку EDIT и обработать событие CBN_SELCHANGE (рис. 8.13).

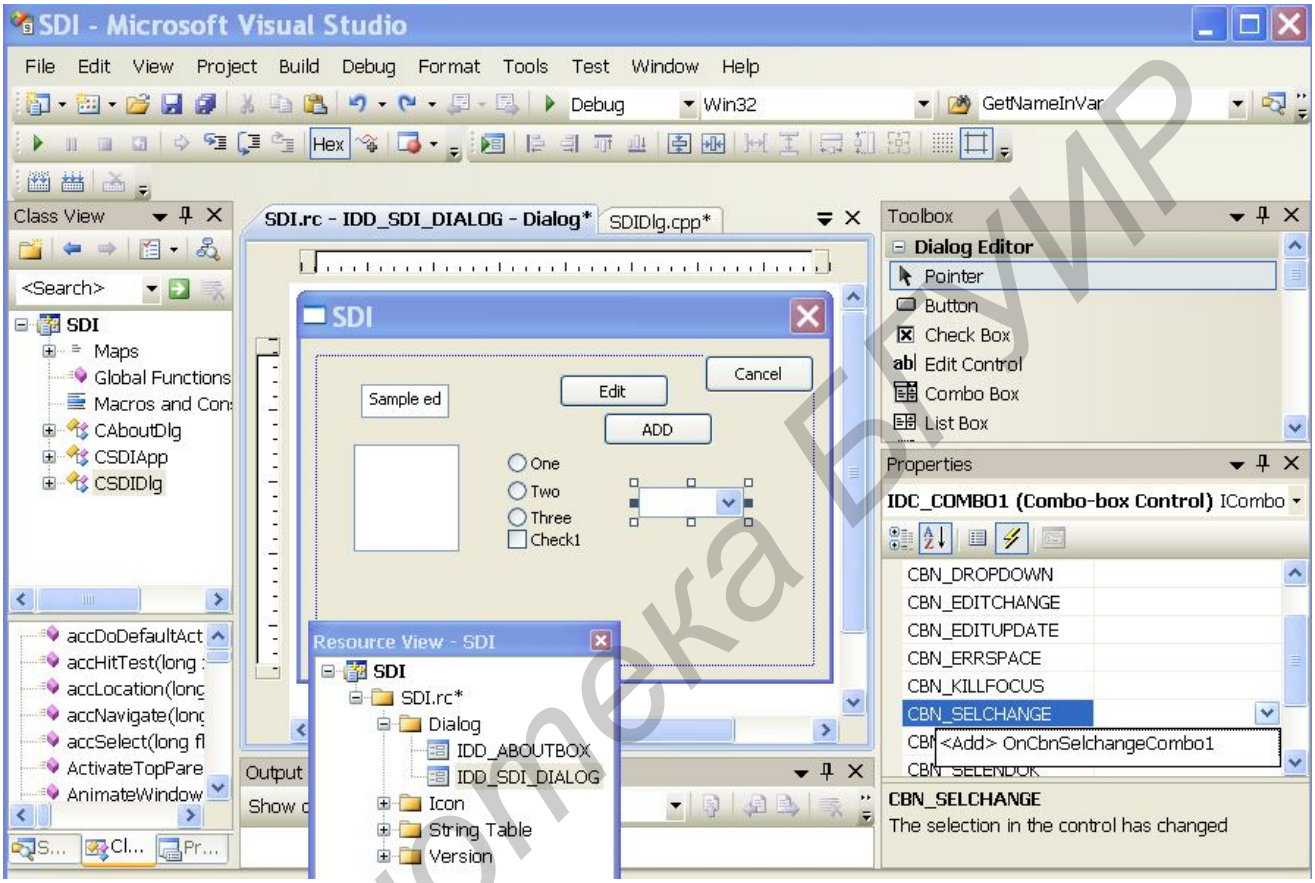

Рис. 8.13. Обработка события CBN_SELCHANGE

Код функции OnSelchangeCombo1() представлен в листинге 8.5.

Листинг 8.5.

```
void CSDIDlg::OnSelchangeCombo1()
{
UpdateData();
m_IndexCombo=
m_combo.GetCurSel();
}
```

Определим новую переменную класса CSDIDlg:

```
public:
int m_IndexCombo;
```

и инициализируем ее в функции CSDIDlg::OnInitDialog():

```
m_IndexCombo= -1;
```

Тогда функция редактирования будет выглядеть так, как представлена в листинге 8.6.

Листинг 8.6.

```
void CSDIDlg::OnEdit()
{
CString str;
UpdateData();
m_combobox.GetWindowText(str);
m_combobox.DeleteString
(m_IndexCombo);
m_combobox.InsertString
(m_IndexCombo, str);
UpdateData(FALSE);
}
```

8.7.2. Удаление, добавление, редактирование элементов списка list box

Добавим на диалог элемент list box. Для вывода выбранного элемента списка list box в edit box можно предусмотреть кнопку IDC_Update. Обработчик события для нее представлен в листинге 8.7.

Листинг 8.7.

```
void CSDIDlg::OnBnClickedUpdate()
{
int i;
UpdateData();
if ((i=m_listbox.GetCurSel())==LB_ERR)
{AfxMessageBox("Не выбран элемент в list");
return false;}
else
{m_listbox.GetText(i,m_edit);
m_IndexList=i;
UpdateData(FALSE);
}
}
```

Эта функция перепишет выбранный в списке элемент в поле редактирования. Можно вывести выбранный элемент list box в edit box путем выполнения двойного щелчка на list box, обработав событие LBN_DBLCLK (рис. 8.14).

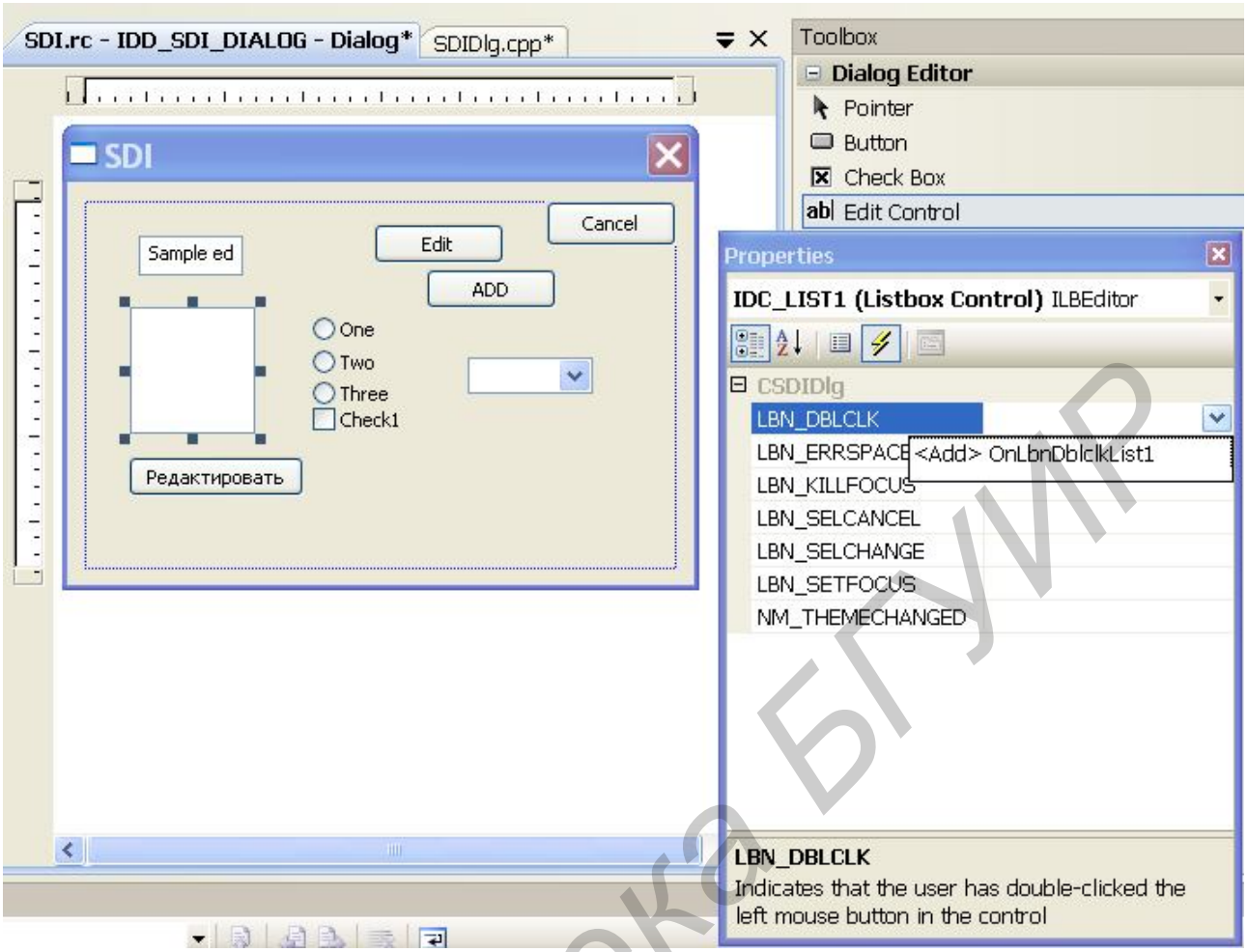

Рис. 8.14. Обработка события LBN_DBLCLK

Реализация обработчика представлена в листинге 8.8.

Листинг 8.8.

```
void CSDIDlg::OnDblclkList1()
{
int i;
UpdateData();
i=m_listbox.GetCurSel();
{m_listbox.GetText(i,m_edit);
m_IndexList=i;
UpdateData(FALSE);
}
}
```

Отредактированный элемент теперь следует вернуть из поля редактирования edit box в list box. Тогда обработчик кнопки завершения редактирования с идентификатором IDC_EnterEdit (рис. 8.15) может выглядеть так, как представлено в листинге 8.9.

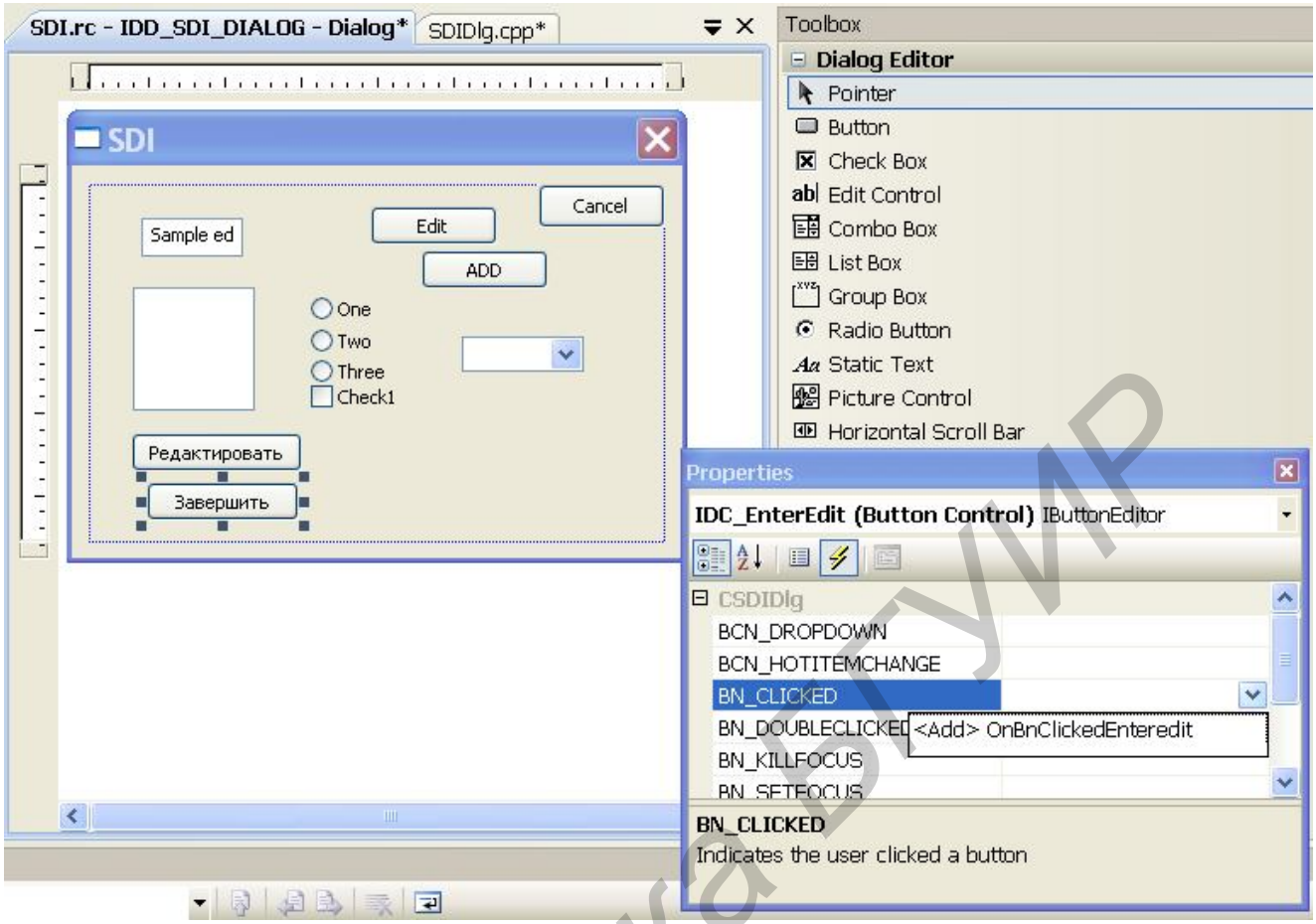

Рис. 8.15. Обработка события BN_CLICKED

Листинг 8.9.

```
void CSDIDlg::OnBnClickedEnteredit()
{
UpdateData();
m_listbox.DeleteString(m_IndexList);
m_listbox.InsertString(m_IndexList,m_edit);
UpdateData(FALSE);
}
```

Обработчик для добавления элемента в list box представлен в листинге 8.10.

Листинг 8.10.

```
void CSDIDlg::OnAdd()
{
UpdateData();
m_listbox.AddString(m_edit);
UpdateData(FALSE);
}
```

8.7.3. Работа с двумя списками

Для переписи одного выделенного элемента из списка combo box в список list box создадим кнопку «MoveOne» и обработаем событие нажатием на нее. Чтобы обойтись одной кнопкой для переписи элемента из первого списка во второй и из второго в первый, можно использовать сообщения CBN_SETFOCUS для combo box и LBN_SETFOCUS для list box (или LBN_SETFOCUS от двух различных list box). Определим в классе CSDIDlg член-переменную m_focus и инициализируем ее в методе OnInitDialog() значением 1. Добавим в проект обработчики сообщений для CBN _SETFOCUS и LBN_SETFOCUS, как показано в листинге 8.11. Обработчик кнопки «MoveOne» представлен в листинге 8.11.

Листинг 8.11.

```
void CSDIDlg::OnSetfocusCombo1()
{
m_focus=1;
}
void CSDIDlg::OnSetfocusList1()
{m_focus=2;
}
void CSDIDialog::OnMoveOne()
{
CString str;
int i;
UpdateData();
switch (m_focus)
{
case 1:
if ((i=m_combobox.GetCurSel())==CB_ERR)
{AfxMessageBox("Не выбран элемент в combo");
break;}
m_combobox.GetLBText(i,str);
m_combobox.DeleteString(i);
m_listbox.AddString(str);
UpdateData(false); break;
case 2:
if ((i=m_listbox.GetCurSel())==LB_ERR)
{AfxMessageBox("Не выбран элемент в list");
     break;}
m_listbox.GetText(i,str);
m_listbox.DeleteString(i);
m_combobox.AddString(str);
UpdateData(false);
     break;
default:
AfxMessageBox("Не выбран элемент");
     break;
}}
```

Можно управлять переписью с помощью радиокнопки, тогда обработчик OnMoveOne() будет выглядеть, как представлено в листинге 8.12.

Листинг 8.12.

```
void CSDIDlg::OnMoveOne()
{
CString str;
int i;
UpdateData();
     switch (m_radio)
{
case 0:
if ((i=m_combobox.GetCurSel())==CB_ERR)
     {AfxMessageBox("Не выбран элемент в combo");
     break;}
m_combobox.GetLBText(i,str);
m_combobox.DeleteString(i);
m_listbox.AddString(str);
UpdateData(false);
     break;
case 1:
if ((i=m_listbox.GetCurSel())==LB_ERR)
     {AfxMessageBox("Не выбран элемент в list");
     break;}
m_listbox.GetText(i,str);
m_listbox.DeleteString(i);
m_combobox.AddString(str);
UpdateData(false);
     break;
default:
AfxMessageBox("Не выбран элемент");
     break;
}}
```

Можно управлять переписью с помощью кнопки-флажка (BOOL m_check), тогда обработчик OnMoveOne() будет выглядеть, как представлено в листинге 8.13.

Листинг 8.13.

```
void CSDIDlg::OnMoveOne()
{
CString str;
int i;
UpdateData();
if (m_check)
{
if ((i=m_combobox.GetCurSel())==CB_ERR)
```

```
{AfxMessageBox("Не выбран элемент в combo");
return true;}
m_combobox.GetLBText(i,str);
m_combobox.DeleteString(i);
m_listbox.AddString(str);
UpdateData(false);
}
else{
if ((i=m_listbox.GetCurSel())==LB_ERR)
     {AfxMessageBox("Не выбран элемент в list");
     return true;}

m_listbox.GetText(i,str);
m_listbox.DeleteString(i);
m_combobox.AddString(str);
UpdateData(false);
}}
```

Для переписи всех элементов списка следует создать кнопку MoveAll и запрограммировать так же, как MoveOne, только предусмотреть цикл для переписи всех элементов списка.

8.7.4. Множественный выбор элементов в списке

Множественный выбор элементов списка возможен только для класса CListbox . При создании ресурса list box необходимо в меню Properties задать selection = Extended. В этом случае множественный выбор будет выполняться мышью при нажатой клавише Shift и сплошным прямоугольником. Если указать selection = Multiple, множественный выбор будет выполняться мышью в произвольном порядке (не обязательно сплошным прямоугольником). Код для обработки кнопки Extended в случае стиля Extended представлен в листинге 8.14.

Листинг 8.14.

```
void CSDIDlg::OnExtended ()
{
//Extended Перепись из List box в Edit box
CString str;
int nIndex;
UpdateData();
m_edit="";
int i=m_listbox.GetCaretIndex( );
// Конец отметки (фокус)
int k=m_listbox.GetAnchorIndex( );
//Начало отметки (анкер)
if (k>i) nIndex=i;
else nIndex=k;
int j= m_listbox.GetSelCount( );
```

```
while (j--){
m_listbox.GetText(nIndex++,str);
str+="\r\n";
m_edit+=str;
}
UpdateData(FALSE);
}
```

Код для обработки кнопки Multi в случае стиля Multiple представлен в листинге 8.15.

Листинг 8.15.

```
void CSDIDlg::OnMulti()
{
// Перепись из List box в Edit box
CString str; int i;
UpdateData();
m_edit="";
int nMaxStrings=200;
int IndexBuf[200];
int nStrings=m_listbox.GetSelItems
(nMaxStrings,IndexBuf);
int k=0;
while (nStrings--){
i=IndexBuf[k];
m_listbox.GetText(i,str);
k++;
str+="\r\n";
m_edit+=str;
}
if (m_edit.GetLength()>100){
AfxMessageBox ("Количество
символов превышает 100");
UpdateData(TRUE);
}
else
UpdateData(FALSE);
}
```

В листинге 8.16 предствлен обработчик кнопки поиска и выделения элементов «Find» в list box со стилем multiple, содержащих фрагмент, заданный в окне редактирования.

Листинг 8.16.

```
void CSDIDlg::OnFind()
{int nIndex, i;
CString str;
UpdateData(TRUE);
str=m_edit;
m_listbox.SetSel( -1,FALSE );
```

```
i=-1;
while (1)
{nIndex=i;
i=m_listbox.FindString(nIndex, str);
if (nIndex<i)
{m_listbox.SetSel(i,true);
}
else break;
}
UpdateData(FALSE);
}
```

Создадим кнопку «Random», по нажатии которой в список list box выводятся случайные числа в отсортированном порядке. Для генерации случайных чисел используется функция

```
time_t time( time_t *timer );
```

которая возвращает системное время – текущее значение счетчика времени в относительных (elapsed) секундах. Описание функции представлено в заголовочном файле <time.h>, time_t – это long integer.

Функция

```
void srand( unsigned int seed );
```

устанавливает начальную точку для получения случайного числа. Описание функции представлено в заголовочном файле <stdlib.h>, seed – начальная точка для генератора случайных чисел. Для реинициализации генератора случайных чисел используется seed = 1.

Функция

```
int rand( void );
```

генерирует псевдослучайное число в диапазоне от 0 до RAND_MAX, описание функции представлено в заголовочном файле <stdlib.h>.

Метод OnRandom() генерирует 100 случайных чисел, сортирует их в убывающем порядке и выводит в элемент управления list box. Элемент управления listbox должен иметь стиль Selection=Multiple, свойство Sort выключено. Код метода представлен в листинге 8.17.

Листинг 8.17.

```
void CSDIDlg::OnRandom()
{int i,j;
UINT Number;
CUIntArray array;
```

```
char str[20];
srand( (unsigned)time( NULL ) );
array.SetSize(100,20);
array.Add(rand());
for( i = 1; i < 100;i++ )
{Number=rand();
for (j=0;j<array.GetSize();j++ ){
if (Number>array.GetAt(j)) {
// Вставить перед элементом, который меньше
array.InsertAt(j,Number);
break;
}
}
// Добавить в конец, если не вставили
if (j==array.GetSize()) array.Add(Number);
 }
// Вывести массив в m_listbox
m_listbox.ResetContent();
for ( j=0; j < array.GetSize();j++){
     wsprintf(str,"%d",array.GetAt(j));
     if (m_listbox.AddString (str)==LB_ERR){
     AfxMessageBox ("Ошибка при добавлении в список");}
     }
UpdateData(false);
}
```

8.7.5. К программированию калькулятора

Выполнение операции сложения чисел, заданных в двух окнах редактирования (m_number1 и m_result), по нажатии кнопки Plus представлено в листинге 8.18.

Листинг 8.18.

```
void CSDIDlg::OnPlus()
{
int num1, result;
UpdateData();
sscanf(m_number1, "%d", &num1 );
sscanf(m_result, "%d", &result );
result+=num1;
m_result.Format("%d", result);
//sprintf (m_result, "%d",result); вариант форматирования строки
m_number1="";
UpdateData(FALSE);
}
```

Задать выполняемую операцию можно различными путями: с помощью нажимаемой кнопки (+, -, =, /, *), радиопереключателя и других элемен-

тов интерфейса. Набор числа с помощью кнопок можно выполнить, как представлено в листинге 8.19.

Листинг 8.19.

```
void CSDIDlg::OnOne()
{ UpdateData();
m_result+="1";
UpdateData(FALSE);
}
void CSDIDialog::OnTwo()
{
m_result+="2";
UpdateData(FALSE);
}
```

8.7.6. Управление состоянием кнопок

Кнопка на диалоговом окне может находиться в различных состояниях, в частности быть или не быть доступна. Пример управления состоянием кнопок представлен в листинге 8.20.

Листинг 8.20.

```
void CSDIDialog::UpdateButtons()
{
UpdateData(true);
if (m_edit!=””) {
GetDlgItem(IDC_ADD)->EnableWindow(true);
GetDlgItem(IDC_UPDATE)->EnableWindow(true);
GetDlgItem(IDC_POISK)->EnableWindow(true);
} else {
GetDlgItem(IDC_ADD)->EnableWindow(false);
GetDlgItem(IDC_UPDATE)->EnableWindow(false);
GetDlgItem(IDC_POISK)->EnableWindow(false);
}
if (m_list.GetSelCount() != 0) {
GetDlgItem(IDC_DELETE)->EnableWindow(true);
} else {
GetDlgItem(IDC_ DELETE)->EnableWindow(false);
}
```

9. Обзор классов окон библиотеки MFC. Стандартные диалоговые панели

Любое стандартное приложение Windows использует различные элементы управления, такие как кнопки, полосы просмотра, редакторы текстов и т. д., реализованные в виде дочерних окон. Хороший обзор классов окон библиотеки MFC представлен в [8].

9.1. Дочерние окна управления

Так как дочерние окна располагаются на поверхности окна родителя, «прилипая» к ним, приложение может создать в любом своем окне несколько элементов управления, которые будут перемещаться вместе с окном-родителем. Для этого достаточно создать нужные дочерние окна, указав их размеры, расположение и некоторые другие атрибуты. После этого приложение может взаимодействовать с элементами управления, передавая или получая от них различные сообщения.

Каждое дочернее окно создается с помощью вызова функции CreateWindow. Оконная процедура родительского окна посылает сообщения дочерним окнам управления, а дочерние окна управления посылают сообщения обратно оконной процедуре.

Дочернее окно управления обрабатывает сообщения мыши и клавиатуры и извещает родительское окно о том, что состояние дочернего окна изменилось. Можно создавать свои собственные дочерние элементы управления, но есть также возможность использовать преимущества нескольких уже определенных классов окна (и оконных процедур), с помощью которых приложение может создавать стандартные дочерние окна управления, присутствующие обычно во всех Windows-приложениях.

Стандартные дочерние окна управления имеют вид кнопок, флажков, окон редактирования, списков, комбинированных списков, строк текста и полос прокрутки. Приложению нет необходимости беспокоиться о логике обработки мыши этими окнами или о логике их отрисовки. Все это делается в Windows. Все, что остается приложению, – это обрабатывать сообщение WM_COMMAND, которыми дочерние окна информируют оконную процедуру о различных событиях.

Стандартные окна управления создаются с помощью редактора ресурсов, с помощью ClassWizard они увязываются с ассоциированными переменными класса родительского окна, в нашем случае класса диалога. Каждому стандартному элементу диалога соответствует соответствующий класс в MFC. Это классы CButton, CEdit, CStatic, CListbox, CCombobox и CScrollbar.

Константы, идентифицирующие стили стандартных элементов управления, определены в заголовочных файлах Windows и имеют соответственно

следующие префиксы: BS_ – button, ES_ – edit, SS_ – static, LBS_ – listbox, CBS_ – combobox и SBS_ – scrollbar.

9.1.1 Сообщения дочерних окон родительскому окну

Если элемент управления изменяет свое состояние, то функция родительского окна получает сообщение WM_COMMAND. Код уведомления – это дополнительный код, который дочернее окно использует для того, чтобы сообщить родительскому окну более точные сведения о сообщении.

Константы, идентифицирующие различные коды уведомления (notification), определены в заголовочных файлах Windows и имеют соответственно следующие префиксы: BN_ – button, EN_ – edit, LBN_ – listbox, CBN_ – combobox и SBN_ – scrollbar.

9.1.2. Сообщения родительского окна дочерним окнам

Родительское окно также может передавать сообщения дочерним окнам, в ответ на которые это дочернее окно будет выполнять различные действия. Константы, идентифицирующие различные сообщения для дочерних окон управления, определены в заголовочных файлах Windows и имеют соответственно следующие префиксы: BM_ – button, EM_ – edit, LB_ – listbox, CB_ – combobox. Для работы с окнами класса CScrollbar применяются специальные Set/Get-функции WinAPI.

9.2. Кнопки различных стилей (класс button)

Родительское окно будет получать от кнопки сообщение WM_COMMAND с кодом уведомления BN_CLICKED. Этим сообщением кнопка информирует родительское окно о проделанных с ней действиях.

Родительские окна могут посылать следующие сообщения кнопкам:

– BM_GETCHECK и BM_SETCHECK – для установки и снятия меток типа включено/выключено флажков-переключателей и радиопереключателей;

– BM_GETSTATE и BM_SETSTATE – для установки состояния нажата/отпущена всех типов кнопок;

– BM_SETSTYLE – для изменения стиля любой кнопки после ее создания.

9.2.1. Нажимаемые кнопки

Нажимаемые кнопки (push buttons) представляют собой прямоугольник, внутри которого находится текст. Нажимаемые кнопки управления используются в основном для запуска немедленного действия без сохранения

какой-либо индикации кнопки типа «включено/выключено». Эти два типа нажимаемых кнопок управления имеют стили, которые называются BS_PUSHBUTTON и BS_DEFPUSHBUTTON (символы «DEF» означают «по умолчанию – default»).

Функционирование кнопок этих двух стилей при использовании их в диалоговых окнах отличается друг от друга. Когда курсор мыши находится на нажимаемой кнопке и левая клавиша мыши нажата, то кнопка перерисовывается так, чтобы выглядеть нажатой. Отпускание клавиши мыши, когда курсор мыши находится на нажимаемой кнопке, приводит к восстановлению облика кнопки и посылке родительскому окну сообщения WM_COMMAND с кодом нотификации BN_CLICKED.

Приложение может имитировать нажатие кнопки, посылая окну сообщение BM_SETSTATE. Также можно послать нажимаемой кнопке сообщение BM_GETSTATE. Дочерняя кнопка управления возвращает текущее состояние – TRUE, если кнопка нажата, и FALSE (или 0), если она в обычном состоянии.

9.2.2. Флажки-переключатели

Флажки (check boxes) представляют собой маленькие квадратные окна с размещенным обычно справа от окна текстом (если при создании кнопки используется стиль BS_LEFTTEXT, то текст окажется слева). Флажки, как правило, действуют как двухпозиционные переключатели: один щелчок вызывает появление контрольной метки (состояние «включено»); другой щелчок приводит к исчезновению этой метки (состояние «выключено»).

В приложениях флажки обычно объединяются, что дает пользователю возможность установить опции. Двумя наиболее используемыми стилями для флажков являются BS_CHECKBOX и BS_AUTOCHECKBOX. При использовании стиля BS_CHECKBOX приложение само должно устанавливать контрольную метку, посылая сообщение BM_SETCHECK. При стиле BS_AUTOCHECKBOX пользователь самостоятельно включает и выключает контрольную метку, когда выбирает окно.

Двумя другими стилями флажков являются BS_3STATE и BS_AUTO3STATE. Как показывают их имена, эти стили могут отображать третье состояние – серый цвет внутри окна флажка. Серый цвет показывает пользователю, что его выбор не определен или не имеет отношения к делу. В этом случае флажок не может быть включен, т. е. он запрещает какой-либо выбор в данный момент. Однако флажок пошлет сообщение родительскому окну, если щелкнуть по нему мышью.

9.2.3. Радиопереключатели

Радиопереключатели (radio buttons, радиокнопки) похожи на флажки, но их форма не квадратная, а круглая. Жирная точка внутри флажка показывает, что переключатель отмечен.

Радиокнопка может иметь стиль окна BS_RADIOBUTTON или BS_AUTORADIOBUTTON, но последний используется только в окнах диалога. В окнах диалога группы радиопереключателей, как правило, используются для индикации нескольких взаимоисключающих опций. В отличие от флажков, если повторно щелкнуть на радиокнопке, то ее состояние не изменится.

9.3. Списки

С помощью класса CListbox можно создавать одноколоночные и многоколоночные списки, имеющие вертикальную (для одноколоночных списков) и горизонтальную (для многоколоночных списков) полосу просмотра.

Список может быть как с одиночным выбором, так и с множественным. Последний позволяет пользователю выбирать более одного пункта списка. Окно списка посылает сообщение WM_COMMAND своему родительскому окну, когда в списке выбирается какой-либо пункт. Родительское окно может определить, какой пункт списка выбран.

Рассмотрим стили окон списков. В окна списка почти всегда включают идентификатор стиля LBS_NOTIFY, что позволяет родительскому окну получать от окна списка сообщение WM_COMMAND. Если приложение желает иметь возможность сортировать элементы списка, ему необходимо использовать в окне списка и другой часто используемый идентификатор стиля – LBS_SORT.

По умолчанию в списке допускается выбор только одного пункта (Single). Если необходимо создать список с возможностью выборки сразу нескольких пунктов (Multiple), то следует использовать стиль LBS_MULTIPLESEL. По умолчанию оконная процедура окна списка выводит только список элементов без какой-либо рамки вокруг. Рамку можно добавить с помощью стиля WS_BORDER. Для прокрутки содержимого с помощью мыши и вертикальной полосы прокрутки следует использовать стиль WS_VSCROLL.

В заголовочных файлах Windows определяется стиль списка, который называется LBS_STANDARD. Стиль LBS_STANDARD включает в себя наиболее часто используемые стили списка. Он определяется как комбинация:

```
(LBS_NOTIFY|LBS_SORT|WS_VSCROLL|WS_BORDER)
```

Можно пользоваться стилями WS_SIZEBOX и WS_CAPTION, которые дают возможность менять размер окна и перемещать его по рабочей области родительского окна. Когда пользователь щелкает мышью над окном списка,

окно списка получает фокус ввода. Если окно списка имеет фокус ввода, то для выбора пунктов можно пользоваться как мышью, так и клавиатурой.

Когда вы работаете со списком, вырабатываются уведомляющие сообщения (notification) от списка. Вы можете посмотреть все сообщения, которые можете обрабатывать для списка, если откроете закладку Messages окна Properties. Для списка List box могут обрабатываться следующие сообщения:

– LBN_ERRSPACE – превышен размер памяти, отведенный для списка;

– LBN_SELCHANGE – изменен текущий выбор (подсветка перемещается по списку);

– LBN_DBLCLK – на пункте списка имел место двойной щелчок мышью. Окно списка посылает коды уведомления LBN_SELCHANGE и LBN_DBLCLK только в том случае, если в стиль дочернего окна включен идентификатор LBS_NOTIFY;

– LBN_KILLFOCUS – потеря фокуса;

– LBN_SELCANCEL – выбор с элемента снимается;

– LBN_SETFOCUS – получение фокуса.

9.4. Комбинированные списки

Комбинированный список является комбинацией списка и однострочного редактора, поэтому для него используются стили, коды извещения и сообщения, аналогичные списку Listbox, а также некоторые сообщения, специфические для поля редактирования Editbox.

При создании списка Combobox указываются специальные стили комбинированного списка, символические имена которых имеют префикс CBS_. Среди всех стилей комбинированного списка можно выделить три базовых (табл. 9.1).

Таблица 9.1

Базовые стили комбинированного списка

Стиль комбинированного списка	Когда list box видимый	Стиль окна редактирования Combo box
Simple	Всегда	Edit
Drop-down	Когда выбрана стрелка	Edit
Drop-down list	Когда выбрана стрелка	Static

Эти стили Combo box задаются при его создании в меню Properties. По умолчанию предлагается Simple. В этом случае список постоянно отображен на экране. В поле ввода, располагающемся сверху, можно вводить любые данные. С помощью мыши можно выделять элемент в прикрепленном окне списка, и он будет попадать в поле редактирования. Кроме того, если в поле

редактирования набрать символы, совпадающие с начальными символами некоторых элементов списка, эти элементы станут первыми в списке.

Стиль Drop-down (раскрывающийся). В поле ввода (edit control) можно вводить любые данные, а щелкнув на стрелку – раскрыть окно списка и выбрать в нем любой элемент. Отмеченный элемент отображается в поле редактирования. Если в поле редактирования набрать символы, совпадающие с начальными символами некоторых элементов списка, эти элементы станут первыми в списке.

Стиль Drop-list (раскрывающийся). В этом комбинированном списке пользователь сможет выбирать элементы только из раскрывающегося списка. Для этого он, раскрыв список, укажет нужную строку или введет первую букву из выделенной строки.

Combo box можно инициализировать в окне Properties. На вкладке Data есть окно Enter Listbox Items. Каждую строку в нем следует заканчивать выбором Ctrl+Enter.

List box можно инициализировать только тогда, когда начинается диалог, например в функции OnInitDialog, которая увязана с сообщением WM_INITDIALOG класса CDialog.

Сообщения от комбинированного списка, посылаемые родительскому окну, имеют символические имена с префиксом CBN_. Для управления списком Combo box используется набор сообщений, аналогичный набору сообщений для списка List box и редактора текста Edit box. В файле windows.h определены сообщения, специально предназначенные для работы со списком Combo box. Символические имена этих сообщений имеют префикс CB_. Многие методы Combo box и List box совпадают. В Combo box нет методов, связанных с множественным выбором.

9.5. Стандартные диалоговые панели

В состав библиотеки MFC входит ряд классов, представляющих стандартные диалоговые панели. Эти классы позволяют легко реализовать такие часто используемые операции, как открытие и сохранение файла, выбор цвета, выбор шрифта и т. д. Все эти классы наследуются от класса CCommonDialog, который, в свою очередь, является производным по отношению к базовому классу CDialog.

Приведем классы стандартных диалоговых панелей и их назначение:

– CColorDialog – панель для выбора цвета;

– CFileDialog – панель выбора файлов для открытия и сохранения на диске;

– CFindReplaceDialog – панель для выполнения операции поиска и замены;

– CFontDialog – панель для выбора шрифта;

– CPrintDialog – панель для вывода документа на печать;

– CPageSetupDialog – панель выбора формата документа;
– COleDialog – панель для управления технологией OLE.

Классы, управляющие стандартными диалоговыми панелями, определены в файле afxdlgs.h. Поэтому при использовании этих классов в приложении необходимо включить этот файл в исходный текст при помощи директивы #include.

9.5.1. Панель выбора цвета

Чтобы отобразить на экране стандартную диалоговую панель выбора цвета, надо создать объект класса CColorDialog, а затем вызвать метод DoModal. Для создания объекта класса CColorDialog используется следующий конструктор:

```
CColorDialog(COLORREF clrInit=0,DWORD dwFlags=0,
CWnd* pParentWnd=NULL);
```

Первый параметр clrInit позволяет указать цвет, выбранный по умолчанию сразу после открытия диалоговой панели. Если параметр не будет указан, в качестве цвета, выбранного по умолчанию, будет использоваться черный цвет.

Параметр dwFlags содержит набор флагов, управляющих диалоговой панелью выбора цвета. При помощи него можно блокировать или разрешать работу некоторых элементов управления диалоговой панели выбора цвета. Если при создании объекта класса CColorDialog не указать параметр dwFlags, тем не менее можно выполнить настройку диалоговой панели, обратившись непосредственно к элементу m_cc данного класса. Параметр dwFlags, указанный в конструкторе, используется для инициализации m_cc. Изменения в элемент m_cc должны быть внесены до того, как панель будет отображаться на экране.

Последний параметр pParentWnd можно использовать, чтобы указать родительское окно диалоговой панели.

Для вывода диалоговой панели выбора цвета на экран необходимо использовать метод DoModal. После отображения панели на экране пользователь может выбрать из нее цвет и нажать кнопки OK или Cancel для подтверждения выбора цвета или отказа от него.

```
CColorDialog dlgColor;
int iResult=dlgColor.DoModal();
```

Когда диалоговая панель закрывается, метод DoModal возвращает значения IDOK или IDCANCEL в зависимости от того, какую кнопку нажал пользователь.

На экране появится стандартная диалоговая панель выбора цвета Color. В верхней половине диалоговой панели расположены 48 прямоугольников, имеющих различные цвета. Они представляют так называемые основные цвета (Basic colors). Можно выбрать один из этих цветов и нажать кнопку OK. После того как диалоговая панель закрыта (метод DoModal завершил свою работу), можно воспользоваться методами класса CColorDialog, чтобы узнать цвета, выбранные пользователем.

Для определения цвета, выбранного пользователем, можно обратиться к методу GetColor класса CColorDialog. Данный метод возвращает значение COLORREF, соответствующее выбранному цвету. При помощи метода GetSavedCustomColors класса CColorDialog можно определить дополнительные цвета, выбранные пользователем в диалоговой панели Color. Этот метод возвращает указатель на массив из 16 элементов типа COLORREF. Каждый элемент массива описывает один дополнительный цвет.

Если пользователю недостаточно основных цветов, представленных в диалоговой панели Color, он может выбрать до 16 дополнительных цветов. Для этого он должен нажать кнопку Define Custom Colors. Диалоговая панель изменит свой внешний вид – появятся дополнительные органы управления, позволяющие выбрать любой из 16 777 216 цветов. Когда цвет выбран, нужно нажать кнопку Add Custom Colors. Выбранный цвет будет добавлен к дополнительным цветам (Custom colors) – один из свободных прямоугольников окрасится соответствующим цветом.

Когда диалоговая панель Color отображается приложением первый раз, все прямоугольники, отображающие дополнительные цвета, имеют белый цвет. Дополнительные цвета, выбранные пользователем, сохраняются во время работы приложения. После перезапуска приложения дополнительные цвета сбрасываются.

9.5.2. Панель выбора файлов

Среди стандартных диалоговых панелей, для которых в библиотеке MFC создан специальный класс, есть панели для работы с файловой системой – Open и Save As. Диалоговая панель Open позволяет выбрать один или несколько файлов и открыть их для дальнейшего использования. Диалоговая панель Save As позволяет выбрать имя файла для записи в него документа.

Для управления диалоговыми панелями Open и Save As предназначен один класс CFileDialog. Рассмотрим конструктор класса CFileDialog более подробно:

```
CFileDialog(BOOL bOpenFileDialog,
     LPCTSTR lpszDefExt=NULL,LPCTSTR lpszFileName=NULL,
     DWORD dwFlags=OFN_HIDEREADONLY|OFN_OVERWRITEPROMPT,
     LPCTSTR lpszFilter=NULL,CWnd* pParentWnd=NULL);
```

Объекты класса CFileDialog представляют диалоговые панели Open или Save As в зависимости от параметра bOpenFileDialog. Если параметр bOpenFileDialog содержит значение TRUE, то создается объект, управляющий диалоговой панелью Open, а если FALSE – диалоговой панелью Save As. Чтобы создать объект класса CFileDialog, представляющий диалоговую панель для открытия файлов (mFileOpen), и объект, представляющий диалоговую панель для сохранения файлов (mFileSaveAs), можно воспользоваться следующими вызовами конструктора класса:

```
CFileDialog mFileOpen(TRUE);  //для панели открытия файлов
CFileDialog mFileSaveAs(FALSE);    //для панели сохранения файла
```

Параметр bOpenFileDialog является единственным обязательным параметром, который необходимо указать. Остальные параметры конструктора класса CFileDialog задают различные режимы работы панели и могут не указываться.

Во многих случаях имена файлов, которые нужно открыть или закрыть, имеют определенное расширение. Параметр lpszDefExt позволяет задать расширение файлов, используемое по умолчанию. То есть если пользователь при определении имени файла не укажет расширение, имени файла автоматически присваивается расширение, принятое по умолчанию. Если при определении свойств диалоговой панели программист присвоит параметру lpszDefExt значение NULL, то расширение файлов должно задаваться пользователем явно.

В некоторых случаях требуется, чтобы диалоговые панели отображались с уже выбранным именем файла. Чтобы указать имя файла, используемое по умолчанию, применяется параметр lpszFileName. Если параметр lpszFileName имеет значение NULL, данная возможность не реализуется.

С помощью флага dwFlags можно изменить внешний вид и некоторые другие характеристики стандартных диалоговых панелей класса CFileDialog. В него можно записать комбинацию флагов, управляющих различными характеристиками этих панелей. Например, флаг OFN_HIDEREADONLY означает, что из диалоговой панели удаляется переключатель «Read Only», а флаг OFN_OVERWRITEPROMPT (используемый для панели Save As) – что необходимо выводить диалоговую панель с предупреждением, если пользователь выбирает для сохранения имя уже существующего файла.

Диалоговые панели выбора файлов обычно имеют список так называемых фильтров, включающих названия типов файлов и расширения имен файлов данного типа. Выбрав фильтр, пользователь указывает, что он желает работать только с файлами определенного типа, имеющими соответствующее расширение. Файлы с другими расширениями в диалоговых панелях не отображаются. Список фильтров можно указать через параметр lpszFilter. Одновременно можно указать несколько фильтров. Каждый фильтр задается двумя строками – строкой, содержащей имя фильтра, и строкой, в которой

перечислены соответствующие ему расширения имен файлов. Если одному типу соответствует несколько расширений, они разделяются символом «;». Строка, содержащая имя фильтра, отделяется от строки с расширениями файлов символом «|». Если используется несколько фильтров, то они также отделяются друг от друга символом «|». Например, в качестве строки, задающей фильтры, можно использовать строку вида

```
char szFilters[] =
" Data (*.dat) |*.dat| Packed Data (*.pac;*.wav) | *.pac; *.wav|
All Files (*.*)| *.* ||";
```

Диалоговые панели, представленные объектами класса CFileDialog, могут иметь или не иметь родительского окна. Чтобы указать родительское окно, нужно передать конструктору CFileDialog указатель на него через параметр pParentWnd.

Создание объекта класса CFileDialog еще не вызывает отображения соответствующей диалоговой панели. Для этого необходимо воспользоваться методом DoModal класса CFileDialog. При вызове метода DoModal для ранее созданного объекта класса CFileDialog на экране открывается соответствующая диалоговая панель.

```
CFileDialog dlgOpen(TRUE);
int iResult=dlgOpen.DoModal();
```

После того как пользователь завершает работу с диалоговой панелью, метод DoModal вернет значение IDOK или IDCANCEL в случае успешного завершения и нуль – в случае возникновения ошибок. Затем можно воспользоваться другими методами класса CFileDialog, чтобы определить имена выбранных файлов:

– GetPathName – определяет полный путь файла;

– GetFileName – определяет имя выбранного файла;

– GetFileExt – определяет расширение имени выбранного файла;

– GetFileTitle – позволяет определить заголовок выбранного файла;

– GetNextPathName – если диалоговая панель позволяет выбрать сразу несколько файлов, то этот метод можно использовать для определения полного пути следующего из выбранных файлов;

– GetReadOnlyPref – позволяет узнать состояние атрибута «только для чтения» (read-only) выбранного файла;

– GetStartPosition – возвращает положение первого элемента из списка имен файлов.

Наиболее важный метод – GetPathName. Он получает полный путь файла, выбранного из диалоговых панелей Open или Save As. Если диалоговая панель позволяет выбрать сразу несколько файлов, тогда метод GetPathName возвращает массив строк, состоящий из нескольких строк, за-

канчивающихся двоичным нулем. Первая из данных строк содержит путь к каталогу, в котором расположены выбранные файлы, остальные строки содержат имена выбранных файлов. Выделение строки, содержащей путь к каталогу, проблем не вызывает, а чтобы получить имена выбранных файлов, необходимо воспользоваться методами GetStartPosition и GetNextPathName.

Метод GetStartPosition возвращает значение pos типа POSITION. Оно предназначено для передачи методу GetNextPathName и получения очередного имени выбранного файла. Если пользователь не выбрал ни одного файла, метод GetStartPosition возвращает значение NULL. Значение, полученное этим методом, следует записать во временную переменную типа POSITION и передать ссылку на нее методу GetNextPathName. Метод GetNextPathName вернет полный путь первого из выбранных в диалоговой панели файлов и изменит значение переменной pos, переданной методу по ссылке. Новое значение pos можно использовать для последующих вызовов метода GetNextPathName и получения путей всех остальных выбранных файлов. Когда метод GetNextPathName вернет имена всех выбранных файлов, в переменную pos записывается значение NULL.

В панелях Open и Save As имеется переключатель «ReadOnly». По умолчанию этот переключатель не отображается. Если есть необходимость воспользоваться этим переключателем, то нужно отказаться от использования флага OFN_HIDEREADONLY.

10. Доступ к базам данных на основе технологии ODBC

10.1. Создание программы, работающей с БД на основе классов ODBC

Базы данных являются наиболее популярными компьютерными приложениями. Они находят применение фактически в любом виде коммерческой деятельности, начиная от создания списков покупателей и заканчивая платежными ведомостями компании.

Современные версии Visual C++ включают классы, базирующиеся на механизмах ODBC (Open Database Connectivity – открытая связь с базами данных), DAO (Data Access Objects – объекты доступа к данным), OLEDB и др. С помощью мастера AppWizard можно создать простое БД-приложение, не написав ни единой строчки текста на языке C++. Более сложные задачи все же потребуют чистого программирования, но в значительно меньшем объеме [8].

Рассмотрим программирование с использованием ODBC-классов Visual C++. В качестве примера создадим БД-приложение, способное не только отображать записи, хранящиеся в базе данных, но и обновлять, добавлять, удалять, сортировать записи, а также выполнять выборку с использованием фильтров.

Создавая с помощью мастера Visual C++ AppWizard программу, работающую с базами данных, получаем в итоге приложение, широко использующее различные классы ODBC из состава библиотеки MFC. Наиболее важными из этих классов являются CDatabase, CRecordset и CRecordView.

Мастер AppWizard автоматически генерирует текст программы, необходимый для создания объекта класса CDatabase. Этот объект обеспечивает связь между создаваемым приложением и источником данных, с которым оно работает. В большинстве случаев использование класса CDatabase в программах, сгенерированных AppWizard, незаметно для программиста. Вся необходимая обработка обеспечивается самой системой управления.

Кроме того, AppWizard генерирует текст программы создания объекта класса CRecordSet, используемого в приложении. Объект класса CRecordSet представляет собой реальные данные, выбранные в настоящий момент из источника данных, а его методы обеспечивают выполнение операций с этими данными. В дальнейшем данные, выбранные программой в текущий момент, будем называть выборкой данных.

И, наконец, объект класса CRecordView в БД-программе занимает место объекта класса представления, с которым постоянно приходится иметь дело в приложениях, созданных с помощью мастера AppWizard. Окно, создаваемое объектом класса CRecordView, подобно диалоговому окну, выполняющему роль средства общения пользователя с приложением. Это диалоговое окно обеспечивает в приложении связь с объектом класса CRecordSet, осуществляя обмен информацией между программой, элементами управления окна и выбор-

кой данных. Когда с помощью мастера AppWizard создается новое БД-приложение, на программиста возлагается обязанность поместить в окно объекта CRecordView элементы управления, способные выполнять ввод и редактирование данных. Как правило, это текстовые поля. Такие элементы управления следует связать с полями записей базы данных, которые они представляют.

Чтобы создать простую БД-программу, использующую классы ODBC, необходимо выполнить следующие операции:

– зарегистрировать базу данных в операционной системе;

– используя мастер AppWizard, создать заготовку БД-приложения;

– добавить в заготовку приложения программный код, реализующий функции, которые AppWizard автоматически не формирует.

Рассмотрим это на примере создания приложения MyDB, предназначенного для просмотра, добавления, удаления, обновления и сортировки записей таблицы Users, входящей в состав некоторой базы данных пользователей.

10.2. Регистрация базы данных

Создайте на жестком диске папку и расположите в ней файл Users.mdb (источник данных Access). Откройте диалоговое окно ODBC Data Source Administrator (Пуск → Настройка → Панель управления → Администрирование) (рис. 10.1). Выберите «Источники данных ODBC» (рис. 10.2).

Щелкните на кнопке «Добавить». Появится диалоговое окно «Создать новый источник данных». Из списка драйверов выберите Microsoft Access Driver (рис. 10.3), а затем щелкните на кнопке «Готово».

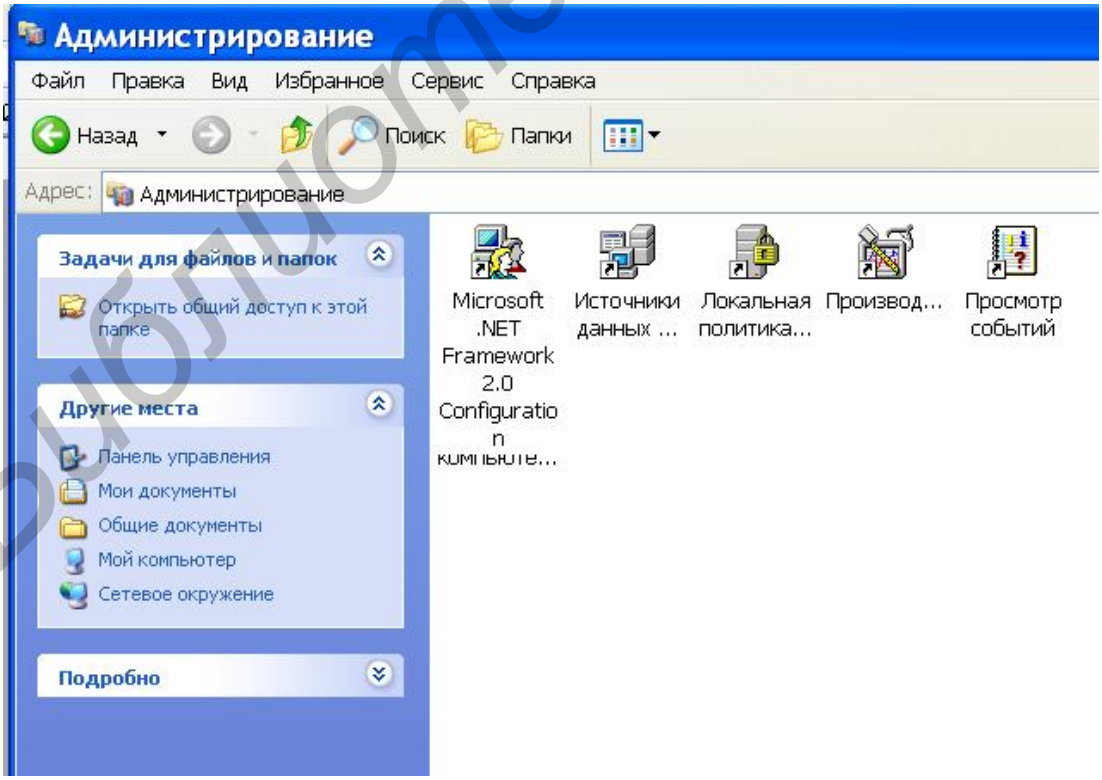

Рис. 10.1. Окно «Администрирование»

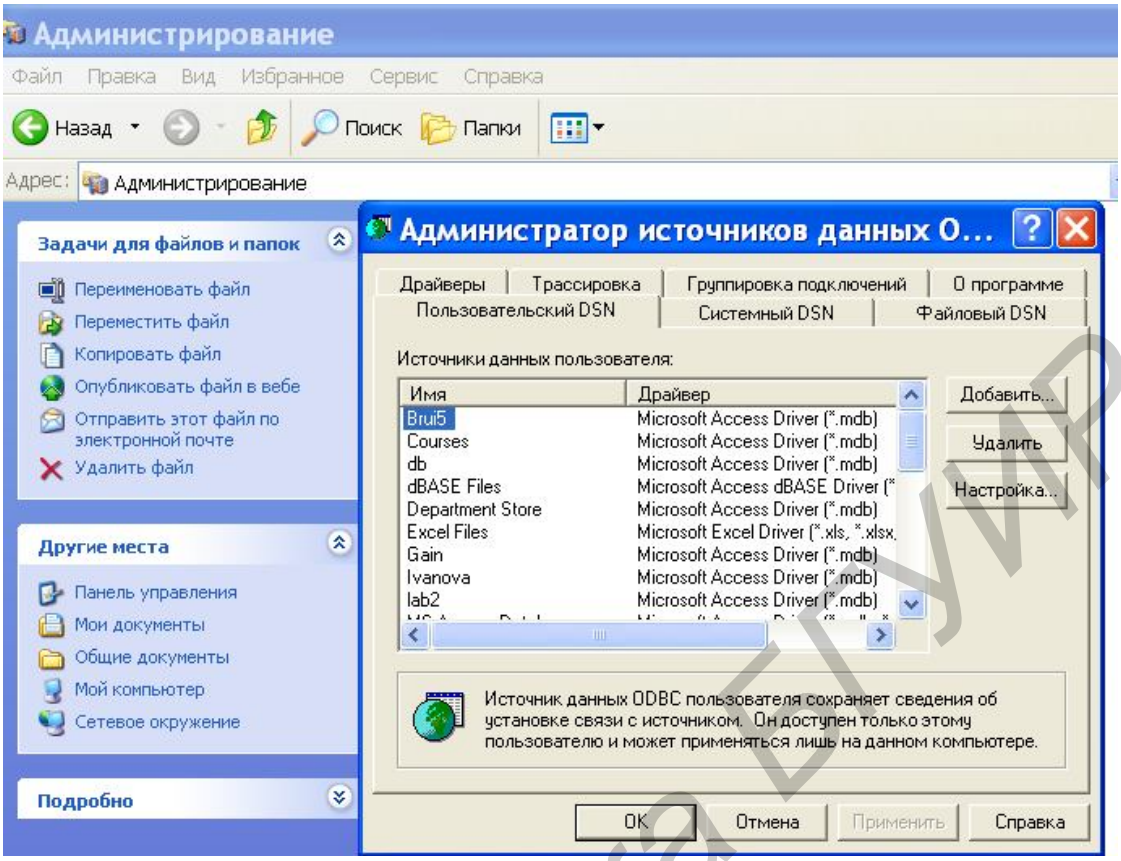

Рис. 10.2. Окно «Администратор источников данных ODBC»

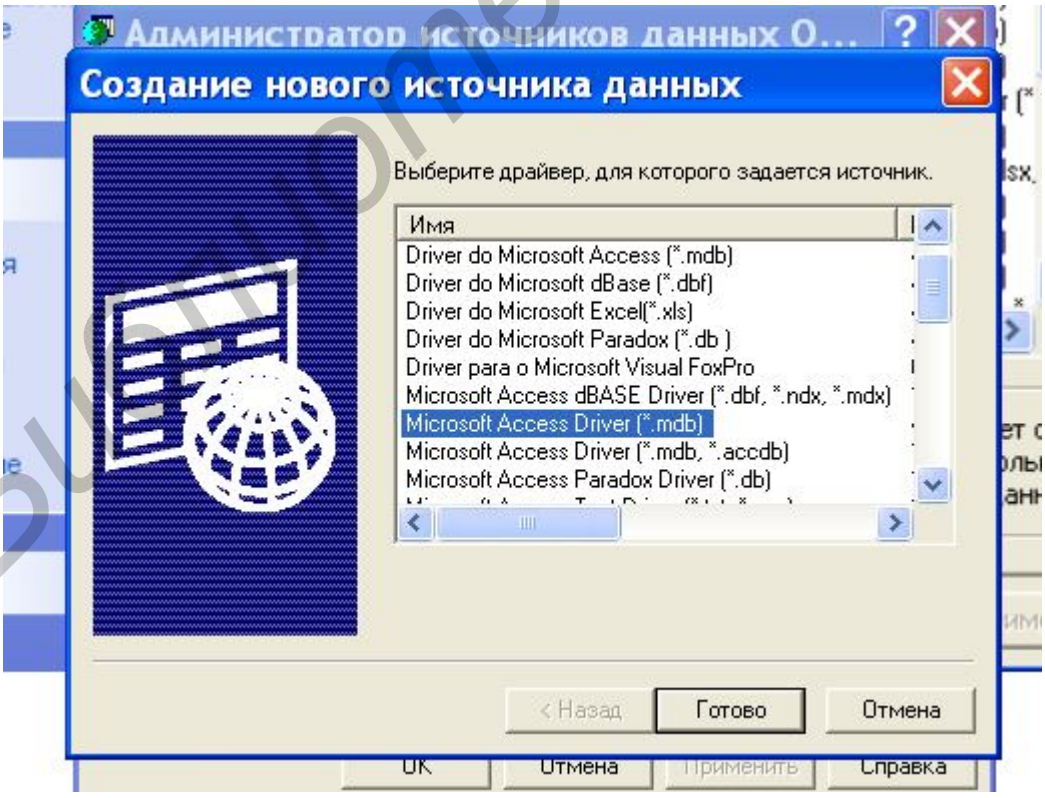

Рис. 10.3. Окно «Создание нового источника данных»

Выберите необходимый источник данных (рис. 10.4).

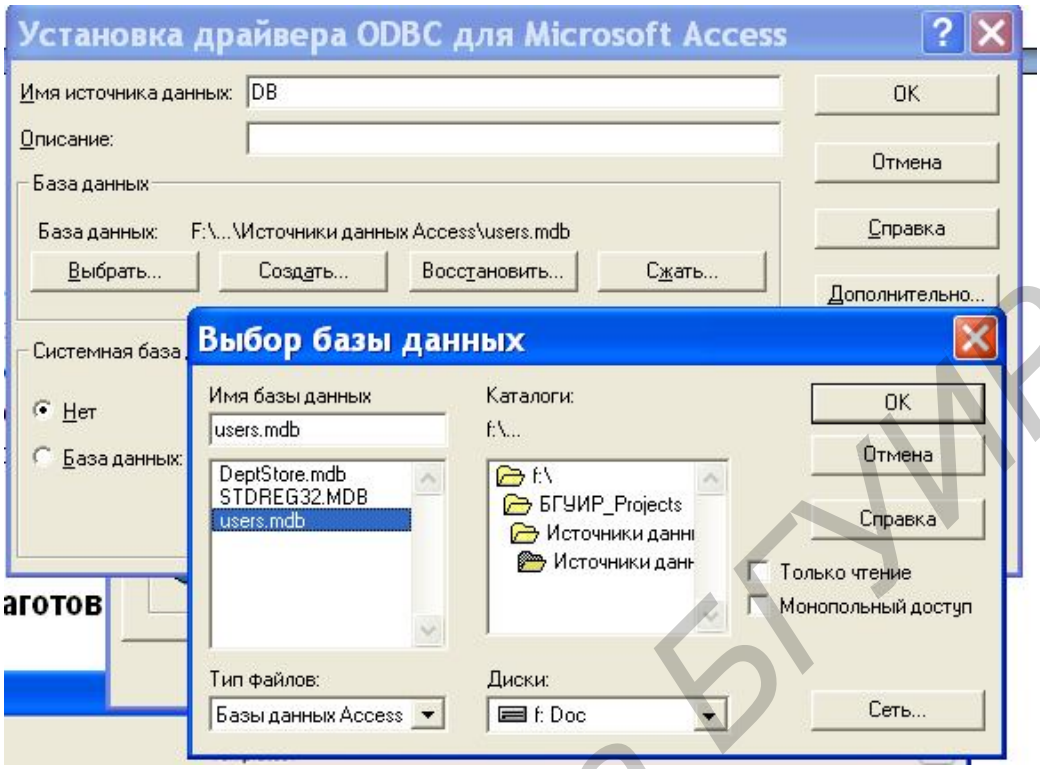

Рис. 10.4. Выбор источника данных users.mdb

Для завершения работы по выбору базы данных щелкните на кнопке OK. Затем в диалоговом окне «Установка драйвера ODBC для Microsoft Access» задайте уникальное имя источника данных (в нашем примере это DB). Теперь в системе установлен доступ к файлу базы данных users.mdb с помощью драйвера ODBC Microsoft Access Driver.

10.3. Создание заготовки для приложения MyDB

Теперь, когда источник данных создан и зарегистрирован, создадим заготовку приложения MyDB. Эта процедура состоит из нескольких этапов.

Выберем тип приложения «MFCApplication», зададим имя проекта (и решения) MyDB, определим папку для проекта (Location). Настроим тип проекта – Single document. Определим уровень поддержки баз данных (Database view with file support) и тип связи с источником данных (ODBC) (рис. 10.5).

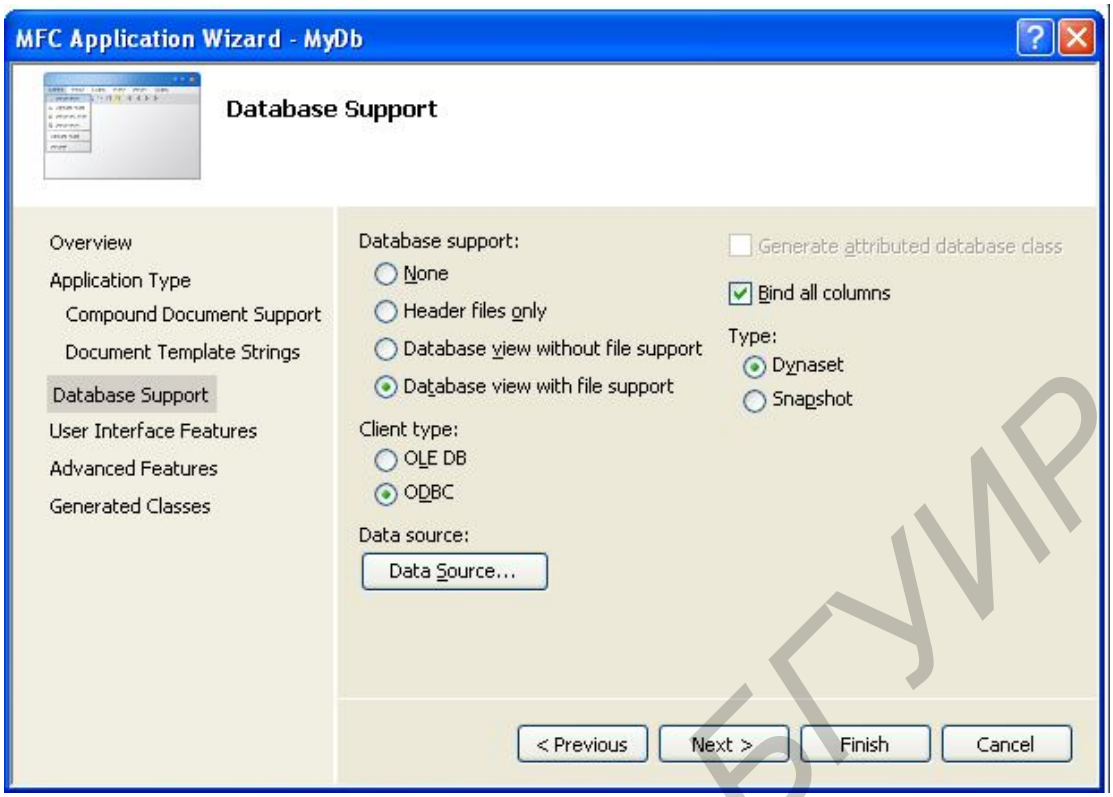

Рис. 10.5. Настройка уровня поддержки баз данных

Выберем зарегистрированный источник данных (DB) (рис. 10.6).

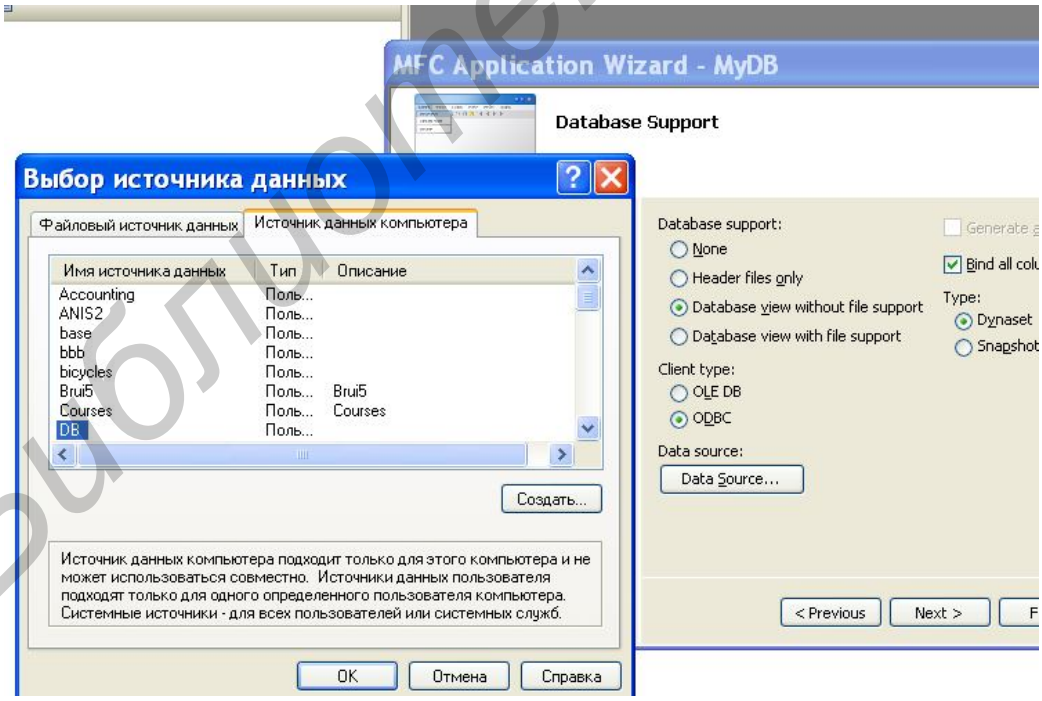

Рис. 10.6. Выбор источника данных

Далее выберем таблицу, с которой будет связан класс выборки – Users (рис. 10.7).

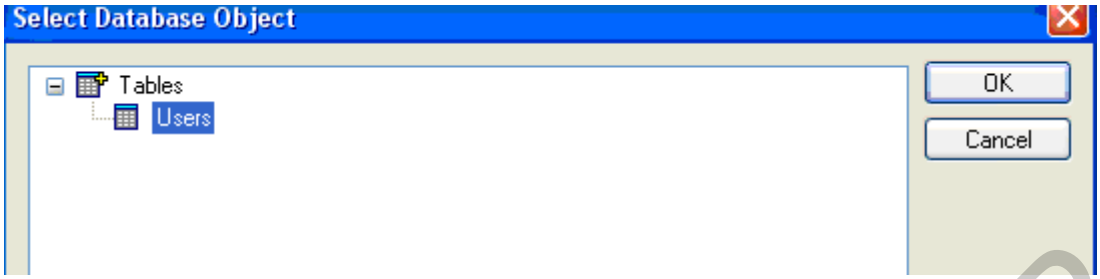

Рис. 10.7. Выбор таблицы

Настройки проекта заверешены. На данный момент уже можно откомпилировать создаваемое приложение. После завершения компиляции можно запустить программу на выполнение. Когда программа будет запущена, раскроется ее окно. На панели инструментов заготовки приложения уже имеются органы управления навигацией по записям таблицы, однако в окне приложения никаких данных нет, поскольку не созданы соответствующие элементы управления и не выполнено их связывание с полями таблицы Users, которые требуется просматривать.

10.4. Создание экранной формы для отображения содержимого базы данных

Следующим шагом в разработке БД-приложения MyDB будет модификация его экранной формы, предназначенной для отображения данных в окне приложения. Поскольку эта форма является просто специализированным типом диалогового окна, модификацию можно легко осуществить с помощью редактора ресурсов Visual Studio.

Для отображения ресурсов приложения щелкните на корешке вкладки ResourceView, который можно открыть View→Other Windows→ResourceView, либо View→ResourceView (рис. 10.8).

Разверните дерево ресурсов, щелкнув на знаке «+»перед папкой MyDb.rc. Окройте папку ресурсов Dialog и сделайте двойной щелчок на идентификаторе диалогового окна IDD_MYDB_FORM и тем самым откройте диалог в редакторе ресурсов (рис. 10.9).

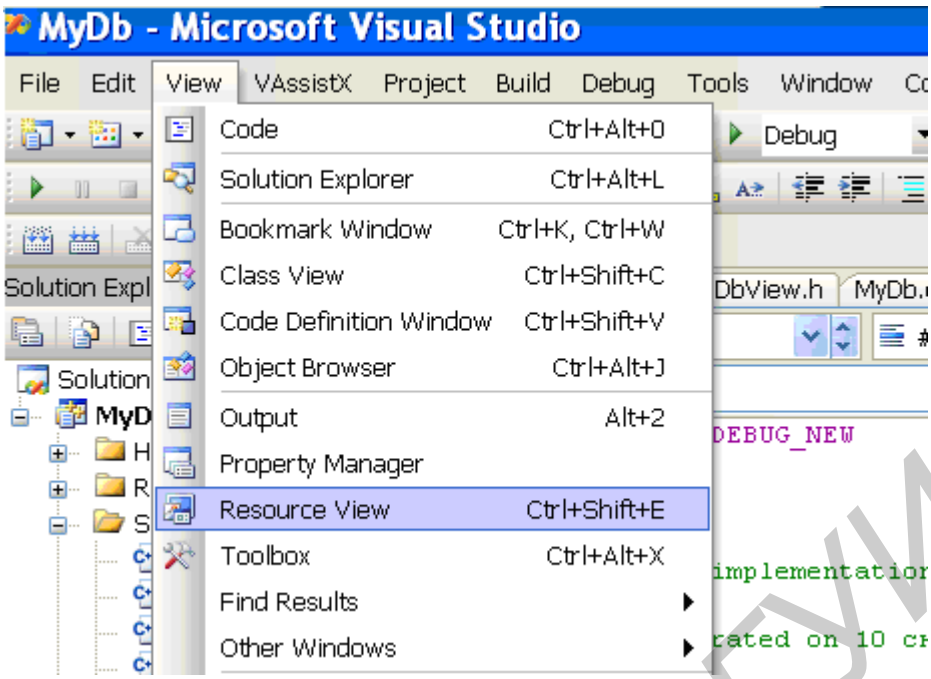

Рис. 10.8. Открытие окна ресурсов

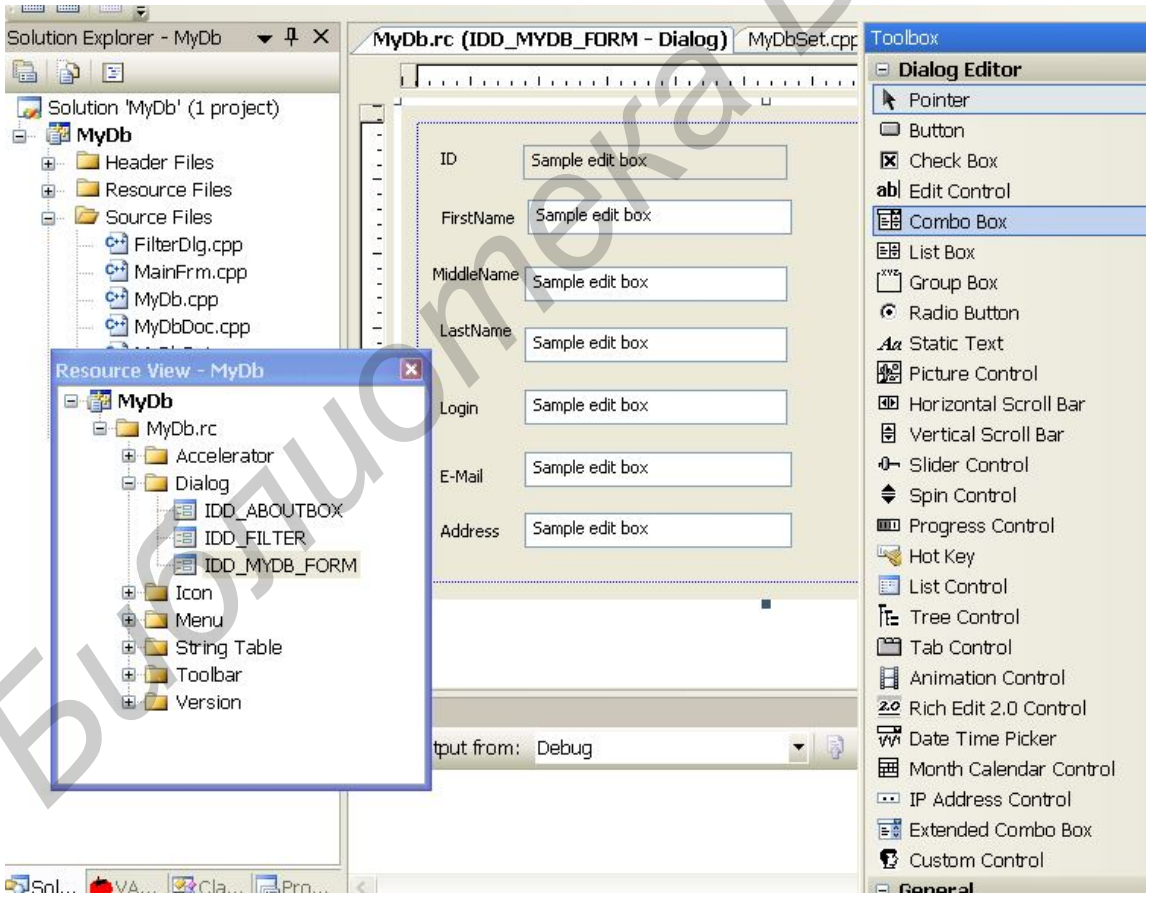

Рис. 10.9. Открытие диалога в окне ресурсов

Пользуясь инструментами редактора диалогового окна, добавьте в него текстовые поля редактирования и статические надписи по образцу. Присвой-

те полям редактирования идентификаторы в соответствии с шаблоном: IDC_название таблицы_название поля (например для поля ID таблицы User IDC_UserID, а для поля FirstName – идентификатор IDC_UserFirstName). Для текстового поля, содержащего идентификатор IDC_UserID, установите свойство Read Only в True (определяется в окне Properties), как показано на рис. 10.10.

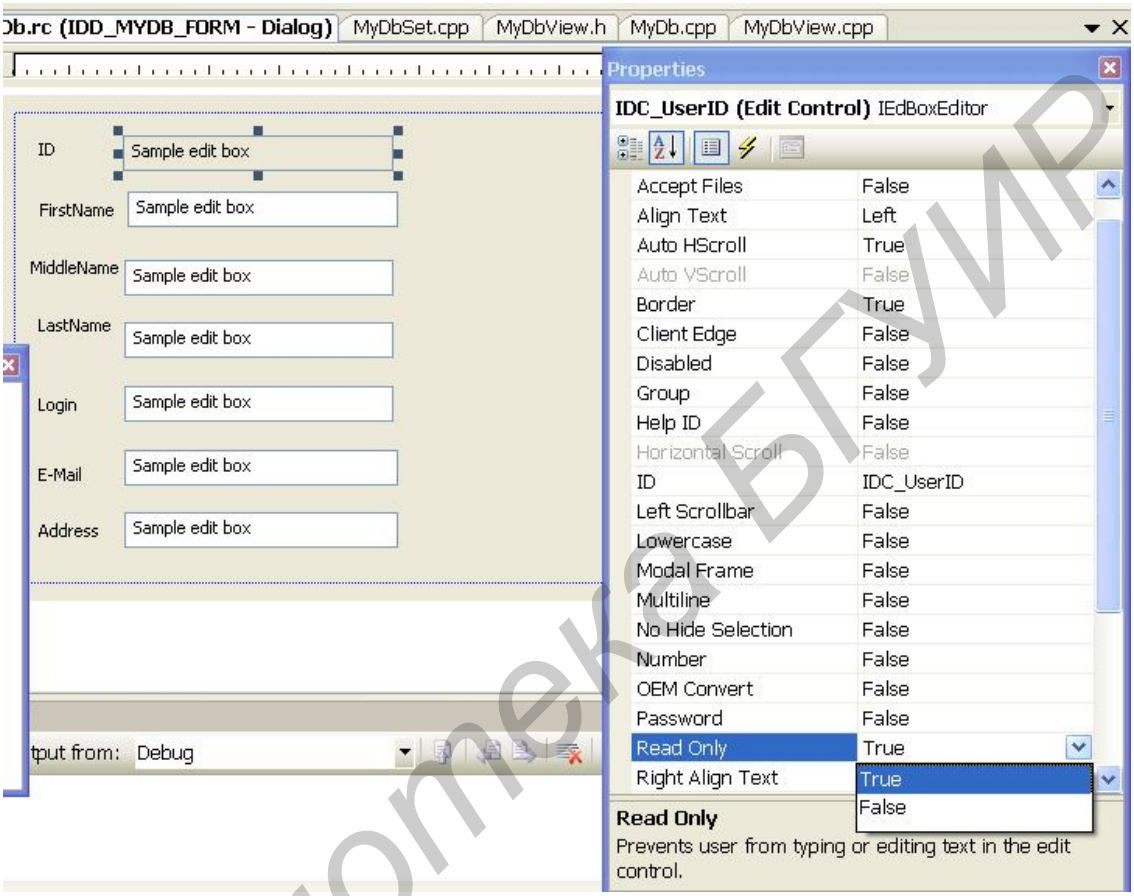

Рис. 10.10. Установка свойства Read Only

Часто перед названием поля на этапе проектирования ставится название таблицы. Особенно это удобно, если поле с таким названием существует в нескольких таблицах. Каждое из этих текстовых полей будет представлять собой поле записи базы данных. Атрибут Read Only (только для чтения) установлен для первого (текстового) поля по той причине, что оно будет содержать первичный ключ базы данных, который не подлежит изменению.

Для создания ассоциированных переменных следует выбрать команду контекстного меню Add Variable каждого Edit box (рис. 10.11).

В раскрывшемся окне Add Member Variable Wizard выберите соответствующий ресурс, например IDC_UserID. Выберите тип переменной (Variable type), категорию (Category), при необходимости права доступа (Access) (рис. 10.12).

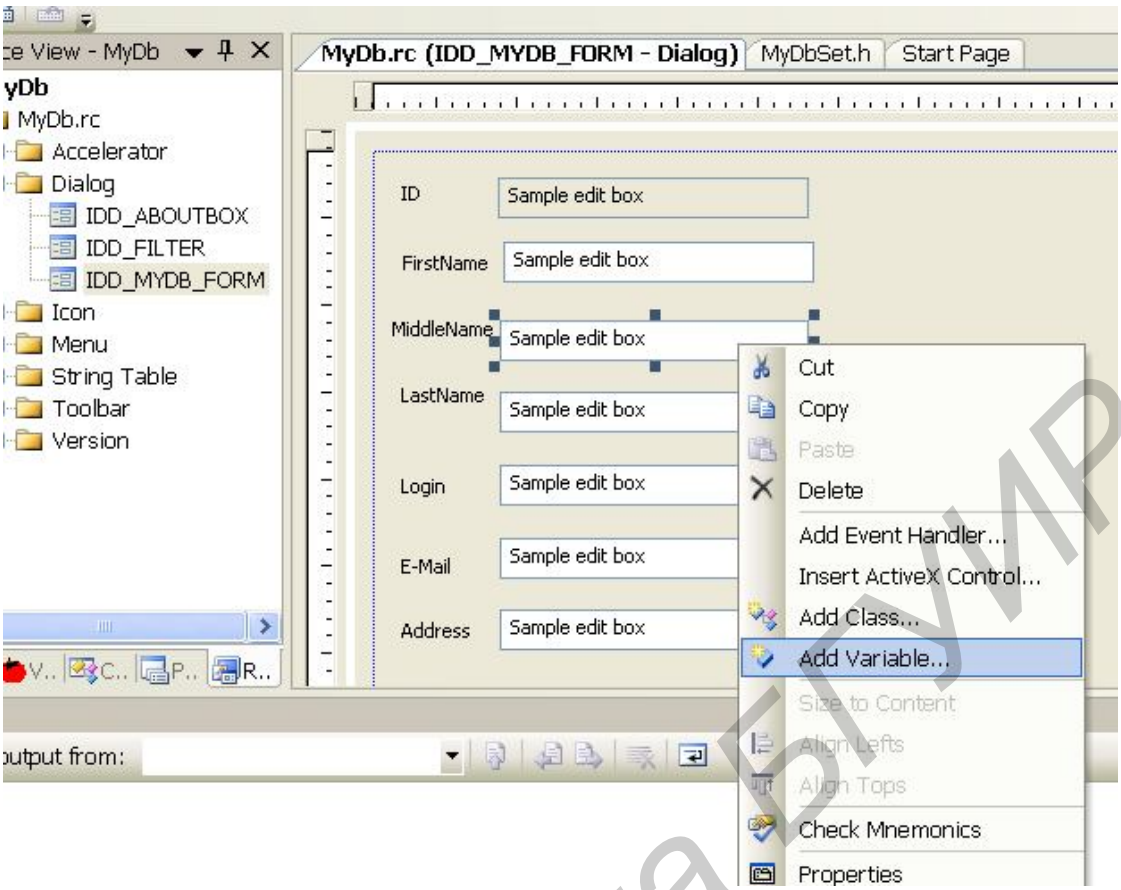

Рис. 10.11. Создание ассоциированных переменных

Рис. 10.12. Добавление ассоциированной переменной

После создания ассоциированных переменных в классе MyDbView.cpp генерируется код, который связывает элементы управления с ассоциированными переменными. Если оттранслировать и запустить программу, отобразится окно (рис. 10.13).

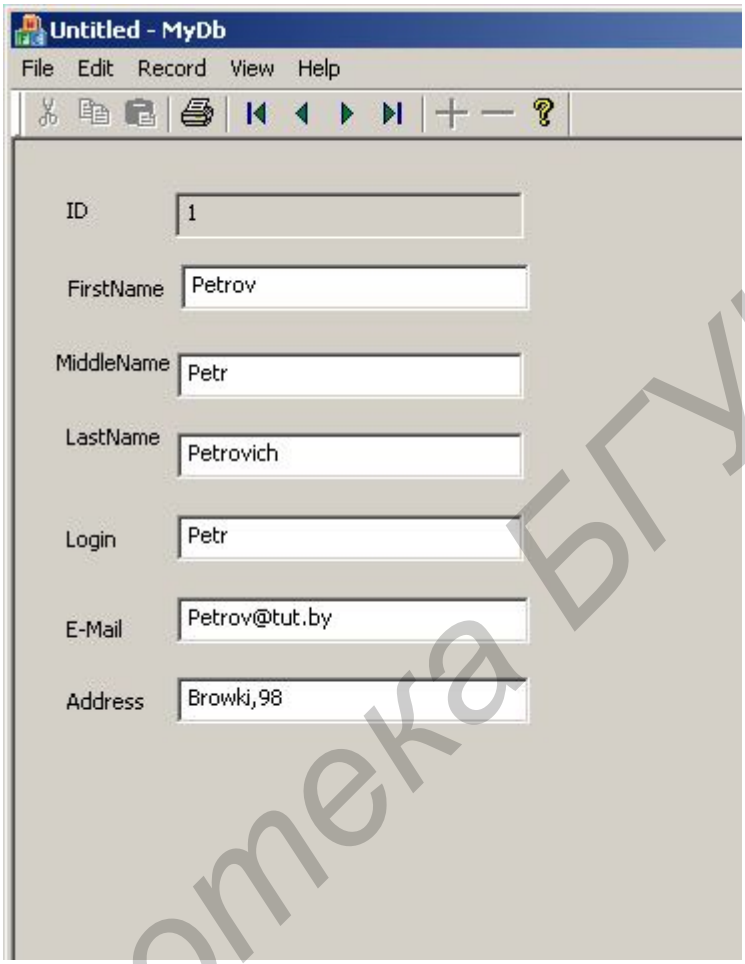

Рис. 10.13. Отображение данных на форме

Теперь наше приложение отображает содержимое записей таблицы Users. Используя элементы управления навигацией, расположенные на панели инструментов приложения, можно перемещаться от одной записи к другой. Можно обновить любую из записей. Для этого достаточно просто изменить содержимое любого из полей записи (за исключением поля ID, которое является первичным ключом и не может быть изменено). При переходе к другой записи приложение автоматически перенесет отредактированные данные в таблицу. Команды меню Record (Запись) приложения позволяют перемещаться по записям в базе данных точно так, как пиктограммы панели инструментов.

Однако возможности приложения MyDB достаточно ограничены. Оно, например, не позволяет добавлять или удалять записи, выполнять сортировку и фильтрацию записей.

10.5. Добавление и удаление записей

Добавление и удаление записей в таблице базы данных реализуется с помощью классов CRecordView и CRecordset, предоставляющих все необходимые методы для выполнения этих стандартных операций.

Необходимо добавить в приложение команды меню Add Record и Delete Record. Далее добавим в проект обработчики событий выбора пунктов меню OnRecordAdd() и OnRecordDelete(). Создадим код для этих обработчиков событий. В объявление класса CMyDBView добавим следующие строки в раздел Attributes:

```
protected:
BOOL m_bAdding;
```

В конструктор класса CMyDBView добавьте следующую строку в конец этой функции:

```
m_bAdding = FALSE;
```

Отредактируйте функцию OnRecordAdd() так, как показано в листинге 10.1.

Листинг 10.1. Функция OnRecordAdd().

```
void CMyDBView::OnRecordAdd()
{
m_pSet->AddNew();
m_bAdding = TRUE;
CEdit* pCtrl = (CEdit*)GetDlgItem(IDC_EMPLOYEE_ID);
int result = pCtrl -> SetReadOnly(FALSE);
UpdateData(FALSE);
}
```

Далее необходимо переопределить виртуальную функцию OnMove в классе CMyDBView. Для этого необходимо вызвать диалог «Add Member Function» (рис. 10.14), и определить виртуальную функцию OnMove() (рис. 10.15).

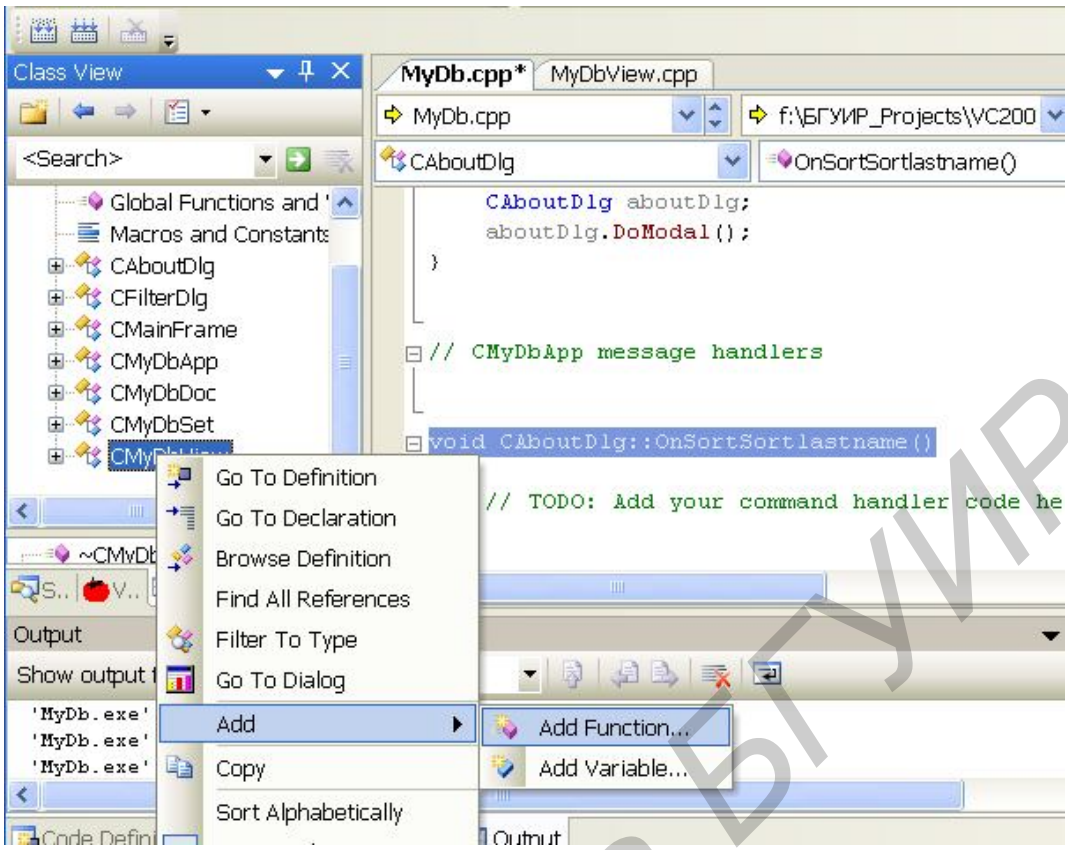

Рис. 10.14. Добавление функции в класс CMyDBView

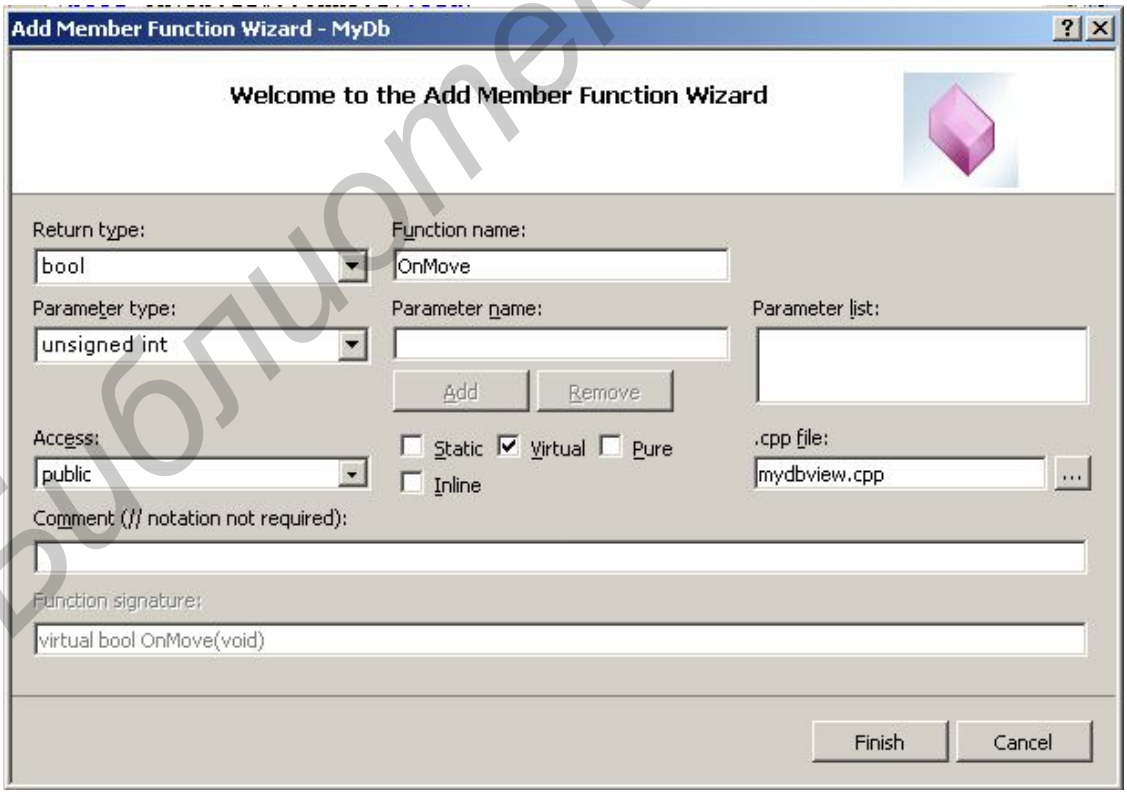

Рис. 10.15. Определение виртуальной функции OnMove()

Отредактируем функцию OnMove() так, чтобы она содержала текст программы, приведенный в листинге 10.2.

Листинг 10.2. Функция OnMove().

```
BOOL CEmployeeView::OnMove(UINT nIDMoveCommand)
{
if (m_bAdding)
     {
     m_bAdding = FALSE;
     UpdateData(TRUE);
     if (m_pSet->CanUpdate())
                    m_pSet->Update();
     m_pSet->Requery();
     UpdateData(FALSE);
     CEdit* pCtrl = (CEdit*)GetDlgItem(IDC_EMPLOYEE_ID);
     pCtrl -> SetReadOnly(TRUE);
     return TRUE;
     }
else
     return CRecordView::OnMove(nIDMoveCommand);
}
```

Отредактируем функцию OnRecordDelete() так, чтобы ее текст соответствовал листингу 10.3.

Листинг 10.3. Функция OnRecordDelete().

```
void CEmployeeView::OnRecordDelete()
{
     m_pSet->Delete();
     m_pSet->MoveNext();
     if (m_pSet->IsEOF())
          m_pSet->MoveLast();
     if (m_pSet->IsBOF())
          m_pSet->SetFieldNull(NULL);
     UpdateData(FALSE);
}
```

Когда приложение начнет работу, на экране раскроется его главное окно с текущей выборкой базы данных. Однако теперь можно добавить в базу данных новую запись, выбрав команду Record→Add Record, либо удалить текущую запись из базы, выбрав команду Records→Delete Record. При добавлении записи приложение отображает в экранной форме пустую запись. Заполните поля новой записи. При переходе к другой записи приложение автоматически внесет новую запись в базу данных. Для того чтобы запись удалить, просто щелкните на пункте меню удаления. Текущая запись (та, которая отображена на экране) исчезнет, и на экран будет выведена следующая запись базы данных.

11. Программирование операций с таблицами базы данных

11.1. Сортировка и фильтрация записей

Часто при работе с базой данных требуется изменить порядок, в котором записи отображаются на экране, или же осуществить поиск записей, удовлетворяющих определенному критерию. Существующие в MFC классы работы с базами данных ODBC располагают методами, позволяющими сортировать выбранные записи по любому из их полей. Кроме того, вызов определенных методов этих классов предоставит возможность ограничить набор отображаемых записей только такими, поля которых содержат указанную информацию, например конкретное имя или идентификатор. Данная операция называется фильтрацией. Добавим функции сортировки и фильтрации в приложение MyDB.

Добавьте пункты меню для сортировки и фильтрации по каждому из полей таблицы (рис. 11.1).

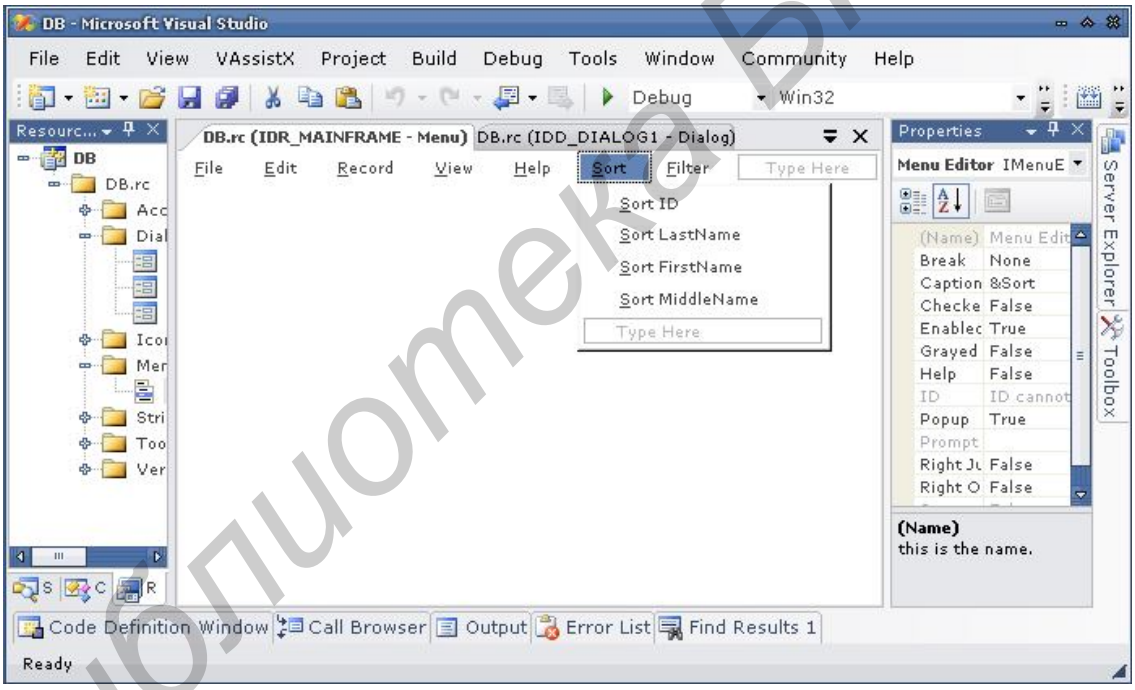

Рис. 11.1. Добавление пунктов меню для сортировки и фильтрации

Для ввода значения параметра фильтрации добавим в проект диалоговое окно IDD_DIALOG1 (рис. 11.2). Расположим на нем элемент управления Edit box с идентификатором IDC_FILTERVALUE (рис. 11.3).

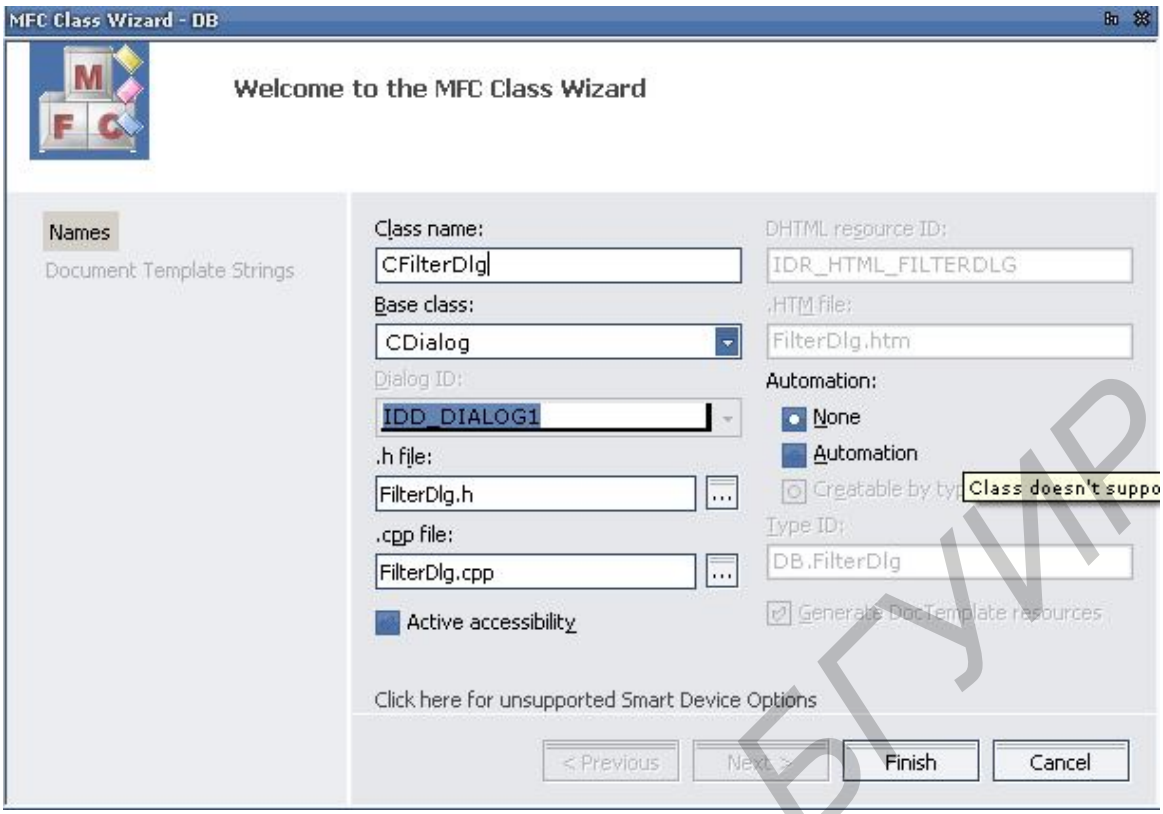

Рис. 11.2. Добавление в проект диалогового окна IDD_DIALOG1

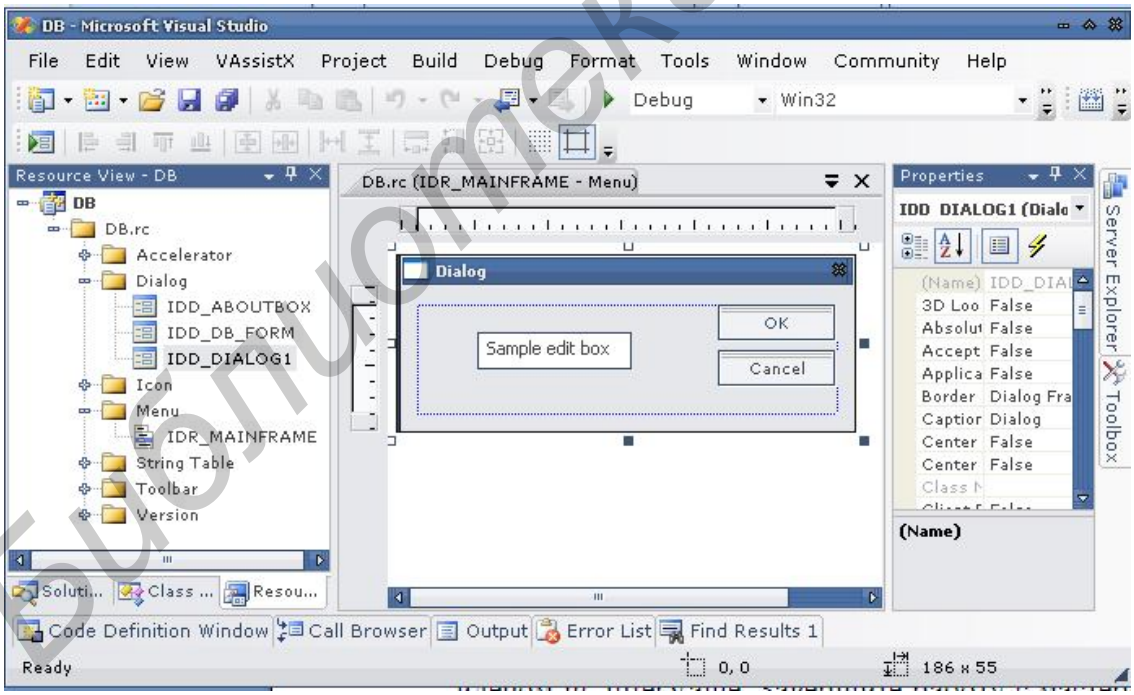

Рис. 11.3. Настройка окна для определения фильтра

Свяжем элемент управления IDC_FILTERVALUE с переменной-членом m_filterValue.

Теперь, когда меню и диалоговые окна уже созданы и связаны с заготовками функций, необходимо добавить в эти заготовки определенный программный код. Отредактируйте текст функции OnSortID() (листинг 11.1).

Листинг 11.1. Функция OnSortID().

```
void CMyDBView::OnSortID()
{
m_pSet->Close();
m_pSet->m_strSort="UserID";
m_pSet->Open();
UpdateData(FALSE);
}
```

Повторите подобную операцию для функций OnSortUserFirstName(), OnSortUserMiddleName() и т. д.

В начало файла MyDBView.cpp после уже имеющихся директив #include добавьте следующую строку:

```
#include "FilterDlg.h"
```

Отредактируйте текст функций OnFilterID(), OnFilterUserFirstName(), OnFilterUserMiddleName(), как показано в листинге 11.2.

Листинг 11.2. Функции OnFilterID(), OnFilterUserFirstName() и др.

```
void CMyDBView::OnFilterID()
{
     DoFilter("UserID");
}

void CMyDBView::OnFilterUserFirstName ()
{
     DoFilter("UserFirstName");
}

void CMyDBView::OnFilterUserMiddleName()
{
     DoFilter("UserMiddleName");
}
```

Все эти функции вызывают функцию DoFilter(). Текст функции, выполняющей фильтрацию записей базы данных, представлен в листинге 11.3.

Листинг 11.3. Функция DoFilter().

```
void CMyDBView::DoFilter(CString col)
{
CFilterDlg dlg;
int result=dlg.DoModal();
if (result==IDOK)
```

```
{
     CString str=col + "=" + dlg.m_filterValue;
     m_pSet->Close();
     m_pSet->m_strFilter=str;
     m_pSet->Open();
     int recCount= m_pSet->GetRecordCount();
     if (recCount==0)
     {    MessageBox("No matching records.");
          m_pSet->Close();
          m_pSet->m_strFilter="";
          m_pSet->Open();
     }
     UpdateData(FALSE);
}}
```

Мы добавили к создаваемому приложению функции сортировки и фильтрации записей базы данных. Теперь можно сортировать записи по любому полю, для чего достаточно просто выбрать имя поля в меню Sort. Кроме того, появилась возможность задать фильтрацию отображаемых записей, выбрав имя требуемого поля в меню Filter, а затем введя значение фильтра в раскрывшемся диалоговом окне Filter. Если значением фильтра является строка, она должна быть взята в одинарные кавычки. Если значением фильтра является число, кавычек не требуется.

11.2. Анализ функции OnSortID()

Все функции сортировки имеют одинаковую структуру. Они закрывают выборку данных, устанавливают свои переменные-члены m_strSort и снова открывают выборку данных, а затем вызывают функцию UpdateData() для обновления окна представления данными из вновь полученной отсортированной выборки данных. Однако в тексте функций сортировки вы не найдете ни одного вызова функции, в названии которой было бы слово Sort. Когда же в таком случае выполняется сортировка? Она выполняется, когда выборка данных открывается заново.

Объект класса CRecordSet (как и объект любого другого класса, производного от CRecordSet, например объект класса CMyDBSet в этой программе) использует специальную строковую переменную m_strSort для определения способа упорядочения записей. Объект анализирует эту строковую переменную при формировании выборки данных и соответственно упорядочивает выбранные из базы записи.

11.3. Анализ функции DoFilter()

Всякий раз, когда пользователь выбирает команду из меню Filter, управляющая программа вызывает соответствующий этой команде метод: OnFilterID(), OnFilterUserFirstName(), OnFilterUserMiddleName(), и т. д. Каж-

дая из этих функций ничего не делает, кроме вызова локального метода DoFilter(), передавая ему в качестве параметра строковую переменную, определяющую поле, по которому требуется выполнить фильтрацию.

Функция DoFilter() независимо от того, какая именно команда была выбрана в меню, всегда отображает одно и то же диалоговое окно, создавая экземпляр объекта класса диалогового окна и вызывая его метод DoModal().

Если значение result не равно IDOK, значит, пользователь выполнил щелчок на кнопке Cancel и весь оператор if пропускается, а функции DoFilter() остается только закончить свою работу.

Внутри конструкции if прежде всего создается строковая переменная, которая будет использоваться для фильтрации записей базы данных. Строковая переменная применяется для выполнения фильтрации записей так же, как это происходит при сортировке. В данном случае строковая переменная называется m_strFilter. Строка, которая используется для фильтрации записей базы данных, должна иметь следующий формат:

ИдентификаторПоля = Значение

Здесь *«ИдентификаторПоля»* является аргументом типа CString функции DoFilter(), а *«Значение»* вводится пользователем в диалоговом окне. Например, если пользователь выберет команду фильтрации по полю UserFirstName и введет в диалоговом окне значение фильтра ‘Скворцов’, функция DoFilter() должна будет создать строку

```
UserFirstName='Скворцов'
```

Сформировав указанную строку, программа будет готова к выполнению фильтрации записей. Для этого, как и в случае сортировки, выборка данных должна быть закрыта, а затем при ее повторном открытии функция DoFilter() выполнит формирование выборки данных с учетом требуемой фильтрации.

Если в результате работы установленного фильтра не будет выбрано ни одной записи, функция DoFilter() обнаружит эту ситуацию, подсчитывая количество записей в создаваемой выборке и сравнивая затем это число с нулем. Если набор записей пуст, программа выводит окно сообщения, информирующее пользователя о сложившейся ситуации. Затем программа закрывает выборку, присваивает строковой переменной фильтра пустое значение и снова открывает выборку записей. Таким образом восстанавливается выборка, включающая все записи таблицы Users. Независимо от того, удалось ли обнаружить записи, отвечающие заданному фильтру, или же выборка данных включает всю базу данных, программа должна заново отобразить данные на экране. Для этого вызывается функция UpdateData().

12. Классы для работы с базами данных

Рассмотрим кратко классы библиотеки MFC, предназначенные для работы с базами данных с использованием механизма ODBC. С подробным обзором этих классов можно познакомиться в [9].

Эти классы, взаимодействуя с другими классами приложения, обеспечивают простой доступ к различным базам данных, использующим драйверы ODBC. Приложениям, которые работают с такими базами данных, следует иметь в своем составе по крайней мере два класса – CDatabase и CRecordSet.

12.1. Класс CDatabase

Объекты этого класса используются для соединения с базами данных, посредством которого можно манипулировать источником данных. Ниже представлен список категорий, на которые можно условно разделить все его компоненты и методы.

Данные. Эти компоненты класса *CDatabase* хранят информацию, используемую в том случае, когда вы хотите работать непосредственно с базой данных, к которой присоединен объект *CDatabase.*

Создание соединения. В эту категорию входят конструктор и методы для открытия/закрытия базы данных.

Атрибуты данных. Сюда относятся десять функций, используемых для получения информации о базе данных, к которой присоединен объект *CDatabase.*

Операции – пять функций, позволяющих обрабатывать транзакции и непосредственно выполнять команды SQL.

Переопределяемые методы. Один метод, позволяющий программисту более конкретно настроить функционирование объекта *CDatabase.*

Конструктор, служит для создания объекта CDatabase:

```
CDatabase::CDatabase(){}
```

После того как объект создан, необходимо установить соединение с определенным источником данных, для чего следует вызвать одну из приведенных ниже функций.

```
virtual BOOL CDatabase::Open(LPCTSTR lpszDSN,
BOOL bExclusive = FALSE, BOOL bReadOnly = FALSE, LPCTSTR
lpszConnect = "ODBC;", BOOL bUseCursorLib = TRUE)
```

Параметр lpszDSN определяет имя источника данных, которое должно быть зарегистрировано с помощью программы ODBC Administrator. Это значение не должно быть равно NULL, если DSN (Data Source Name, Имя источника данных) определено в строке lpszConnect, или может быть равно

NULL, если необходимо предоставить пользователю блок диалога для выбора источника данных.

Параметр bExclusive равен FALSE, если источник данных открывается для совместного использования, иначе он равен TRUE.

Параметр bReadOnly позволяет установить соединение с источником данных в режиме «только для чтения» (TRUE), что приводит к запрещению его обновления. После установления такого соединения все зависимые результирующие множества наследуют этот атрибут.

Параметр lpszConnect определяет строку, описывающую соединение, которая содержит информацию об источнике данных, идентификаторе пользователя, имеющего к нему доступ, пароль, если он требуется источнику данных, и другую информацию. Для совместимости с будущими версиями требуется, чтобы эта строка начиналась с подстроки "ODBC;", указывающей на то, что соединение устанавливается с источником данных ODBC.

Параметр bUseCursorLib указывает на необходимость (TRUE) или необязательность (FALSE) загрузки динамической библиотеки ODBC Cursor Library, позволяющей работать с курсорами базы данных.

Упрощенная версия рассмотренной функции имеет вид

```
virtual BOOL CDatabase::OpenEx (LPCTSTR lpszConnectString,
DWORD dwOptions = 0)
```

Параметр lpszConnectString определяет строку соединения с источником данных ODBC, которая включает его имя, а также дополнительную необязательную информацию, такую как идентификатор и пароль пользователя, например «DSN=Publisher;UID=sa;PWD=irishka». Если в качестве параметра передается NULL, то выводится блок диалога Data Source, в котором пользователь может выбрать источник данных. Параметр dwOptions – битовая маска, которая определяет комбинацию значений, представленных в табл. 12.1.

Таблица 12.1

Значения параметра dwOptions

Флаг битовой маски	Значение
CDatabase::openExclusive	Источник данных открывается для совместного использования
CDatabase::openReadOnly	Источник данных открывается в режиме «только для чтения»
CDatabase::openUseCursorLib	Указывает на необходимость загрузки динамической библиотеки ODBC Cursor Library, позволяющей работать с курсорами
CDatabase::noOdbcDialog	Не выводить блок диалога
CDatabase::forceOdbcDialog	Всегда выводить блок диалога соединения

Значение, заданное по умолчанию (0), означает, что база данных открывается для совместного использования с доступом для записи, динамическая библиотека поддержки курсора не загружается и блок диалога для выбора источника данных отображается только в том случае, если не указана дополнительная информация о соединении.

Обе функции выполняют одну и ту же задачу – установить соединение с источником данных. Разница заключается только в способе задания параметров. Пример открытия соединения с базой данных представлен в листинге 12.1.

Листинг 12.1.

```
CDatabase m_dbSamp; // Создаем объект класса CDatabase
//Открываем соединение с источником данных, указав имя
// источника и идентификатор пользователя (без пароля)
m_dbsamp.Open(_T(«Samples»), FALSE, FALSE, _T(«ODBC;UID=sa»));
//или запрашиваем всю информацию у пользователя
m_dbsamp.Open(NULL);
//Закрываем текущее соединение
…
m_dbsamp.Close();
…
//и открываем новое
m_dbPubl.OpenEx (_T("DSN=Authors; UID=sa"),
```

12.2. Класс CRecordset

Все компоненты и методы этого класса можно разбить на семь категорий.

Компоненты данных служат для хранения информации, используемой для непосредственной работы с базой данных, к которой объект этого класса был присоединен.

Конструирование. В эту категорию входят конструктор и методы для открытия/закрытия форм базы данных.

Атрибуты результирующего набора – функции, используемые для получения информации о результирующем наборе, к которому присоединен объект класса CRecordset.

Операции обновления результирующего набора – четыре операции, предназначенные для обработки транзакций.

Операции перемещения по результирующему набору – функции, позволяющие перемещаться по записям результирующего набора.

Другие операции над результирующим набором – восемь функций, предоставляющих дополнительные функциональные возможности.

Переопределяемые методы – пять переопределяемых функций, позволяющих программисту настроить функционирование объекта класса CRecorset.

В классе CRecordset определены следующие основные компоненты данных:

`UINT CRecordset::m_nFields`

– содержит число полей данных в результирующем наборе (число столбцов, получаемых из источника данных). Это поле должно быть корректно инициализировано в конструкторе класса CRecordset;

`UINT CRecordset::m_nParams`

– содержит число параметров в результирующем наборе (число параметров, посылаемых в параметризированном запросе результирующего набора). Если эта переменная используется, то она должна быть корректно инициализирована в конструкторе. По умолчанию инициализируется нулем;

`CDatabase CRecordset::m_pDatabase`

– содержит указатель на объект класса CDatabase, посредством которого результирующий набор соединяется с источником данных. Эта переменная устанавливается двумя способами: если вы уже установили соединение с источником данных, то при создании объекта класса CRecordset передайте туда указатель на объект класса CDatabase или можно передать туда NULL – при этом CRecordset сам создаст объект CDatabase и соединится с ним. В обоих случаях указатель на базу данных хранится в этой переменной m_pDatabase. И хотя обычно нет необходимости отслеживать ее состояние, все же есть случаи, когда делать это необходимо, например при запуске транзакции или непосредственного выполнения оператора SQL;

`CString CRecordset::m_strFilter`

– используется в качестве фильтра, что позволяет выбирать только записи, удовлетворяющие заданному критерию. Для осуществления «фильтрации» эту переменную следует определять после создания объекта класса CRecordset, но до вызова функции Open, или же «перечитать» запрос с помощью функции Requery. Эта строка служит для определения предложения WHERE оператора SQL. Ключевое слово WHERE не нужно включать в строку фильтра, поскольку библиотека подставляет его сама;

`CString CRecordset::m_strSort`

– содержит имя поля, по которому сортируются записи. Для того чтобы осуществить «сортировку», эту переменную следует определять после создания объекта класса CRecordset, но до вызова функции Open, или же «перечитать» запрос с помощью функции Requery. Эта строка служит для определения предложения ORDER BY оператора SQL. Не нужно включать в строку

сортировки ключевое слово ORDER BY, поскольку библиотека подставляет его сама.

Помимо перечисления полей, по которым производится сортировка, можно указать также одно из ключевых слов (отдельно для каждого поля), определяющих направление сортировки: ASC – по возрастанию и DESC – по убыванию.

В категорию «Конструирование» входят всего три функции: конструктор, Open и Close, отвечающие за создание объекта, открытие и закрытие соединения с источником данных. Конструктор представлен ниже:

```
CRecordset::CRecordset(CDatabase* pDatabase = NULL)
```

Он служит для создания и инициализации объекта класса CRecordSet. В качестве параметра в конструктор можно передать либо указатель на открытую базу данных (параметр pDatabase), либо NULL, что говорит о необходимости открытия базы данных по умолчанию. При создании производного от CRecordset класса у него должен быть только один конструктор, по параметрам совпадающий с базовым.

Функция открытия выборки:

```
virtual BOOL CRecordset::Open (UINT nOpenType =
AFX_DB_USE_DEFAULT_TYPE, LPCTSTR lpszSQL = NULL,
DWORD dwOptions = none)
```

При успешном выполнении функции возвращается ненулевое значение и 0 – в противном случае. Если полученный результирующий набор не пустой, то текущей является первая запись.

Параметр lpszSQL – указатель на строку, содержащую одно из значений: NULL, имя таблицы, оператор SQL, не обязательно с предложениями WHERE или ORDER BY, или оператор CALL, определяющий имя предопределенного запроса или сохраненной процедуры. Порядок столбцов в результирующем наборе должен совпадать с порядком вызова в переопределенной функции DoFieldExchange.

Функцию CRecordset::Open следует вызывать для выполнения запроса, определяющего результирующий набор. Очевидно, что перед ее вызовом объект класса CRecordset должен быть создан. Более того, соединение с источником данных зависит от того, как именно он создан:

– если в качестве параметра был использован указатель на объект CDatabase, который еще не соединен с источником данных, то функция Open использует метод GetDefaultConnect для открытия объекта базы данных;

– если в качестве параметра использовался NULL, то автоматически создается объект класса CDatabase и осуществляется попытка соединения с базой данных.

Доступ к источнику данных посредством объекта CRecordset никогда не может быть эксклюзивным.

При вызове функции Open выполняется запрос, в результате чего выбираются записи на основе критериев, перечисленных в таблице 12.2.

При написании строки SQL необходимо тщательно следить за тем, чтобы в нее не попали дополнительные пробелы. Например, если между круглой скобкой и ключевым словом CALL (а также SELECT) будет хоть один пробел, то библиотека MFC неправильно поймет имя таблицы, что приведет к обработке исключения (исключительной ситуации). Аналогичная ситуация возникает, если пробел появится между круглой скобкой и символом '?' в параметризованном запросе.

Таблица 12.2

Критерии выбора записей

Значение параметра lpszSQL	**Выбираемые записи определяются**	**Пример**
NULL	Строкой, возвращаемой функцией GetDefaultSQL	
Имя таблицы	Столбцами, указанными в функции DoFieldExchange	"Products"
Имя предопределенного запроса (сохраненной процедуры)	Столбцами, определенными в запросе	"(callProductList)"
SELECT список-столбцов FROM список-таблиц	Заданными столбцами из определенных таблиц	SELECT ProductName, UnitPrice FROM Products

Перед вызовом функции Open можно определить дополнительные условия, соответствующие параметрам предложений WHERE и ORDER BY оператора SQL, используя для этого описанные выше переменные класса m_strFilter и m_strSort. Если же их определить после того, как результирующий набор открыт, то они не окажут никакого влияния на результат выполнения запроса. В этом случае необходимо будет воспользоваться функцией Requery для обновления записей.

Функция

```
virtual void CRecordset::Close()
```

– используется для закрытия результирующего набора. Если он не был открыт, то функция просто завершается. После ее выполнения вся память, выделенная для результирующего набора, и дескриптор HSTMT ODBC, ассоциированный с ним, возвращается системе. В том случае, когда объект

класса CRecordset создает с помощью оператора new, вызов функции Close осуществляется автоматически при удалении объекта.

12.3. Класс CRecordView

Объекты этого класса предоставляют для изображения записей базы данных в элементах управления форму, которая непосредственно соединена с объектом CRecordset. Объекты CRecordView используют механизм DDX (Dialog Data Exchange, Обмен данными с блоком диалога) и RFX (Record Field Exchange, Обмен полями записей) для автоматического перемещения данных между элементами управления формы и полями результирующего набора. Кроме того, можно воспользоваться реализованными возможностями перемещения по записям и обновления текущей записи. Все компоненты и методы этого класса можно условно разбить на три категории.

Создание объекта. Имеется только конструктор для создания объекта.

Атрибуты данных – три функции, используемые для получения информации о результирующем наборе, к которому присоединена форма.

Операции – единственная функция, позволяющая программисту изменить указатель на текущую запись.

В категорию «Создание объекта» входит только конструктор, имеющий две реализации:

```
CRecordView::CRecordView(LPCSTR lpszTemplateName)
CRecordView::CRecordView{UINT nIDTemplate)
```

Конструктор создает объект класса. В качестве параметра конструктор принимает идентификатор шаблона блока диалога, задаваемый либо строкой (lpszTemplateName), либо номером (nIDTemplate). При создании класса, производного от CRecordView, в нем можно определить только один конструктор, в котором необходимо вызвать конструктор базового класса CRecordView::CRecordView с идентификатором ресурса в качестве параметра, как это показано в приведенном ниже фрагменте:

```
CDBView::CDBView():CRecordView(CDBView::IDD)
{
//{{AFX_DATA_INIT (CDBView)
m_pSet = NULL;
m_strPrice =_T("");
//}}AFX_DATA_INIT
m_bAdd = FALSE;
m_nSort = ID_SORT_TITLE;
}
```

Функции категории «Атрибуты данных» позволяют получить информацию о представлении записи.

```
virtual CRecordset* CRecordView::OnGetRecordset( )
```

Функция OnGetRecordset() возвращает указатель на объект CRecordset, ассоциированный с формой, позволяя тем самым работать с некоторым результирующим набором. Это чисто виртуальная функция, которая требует обязательного переопределения, и библиотека не может знать, с каким результирующим набором, т. е. объектом класса CRecordset, вы работаете.

Ниже показан фрагмент кода, который создает мастер AppWizard при создании приложения для работы с базой данных:

```
CRecordset* CDBView::OnGetRecordset()
{
return m_pSet;
}
```

Функция IsOnFirstRecord( ):

```
BOOL CRecordView::IsOnFirstRecord( )
```

позволяет определить, является ли текущая запись первой в результирующем наборе, ассоциированном с данной формой. Когда пользователь перемещается за первую запись, библиотека блокирует доступ к элементам пользовательского интерфейса для перемещения на первую и предыдущую запись.

Функция

```
BOOL CRecordView::IsOnLastRecord( )
```

позволяет определить, является ли текущая запись последней в результирующем наборе, ассоциированном с данной формой. Когда пользователь перемещается за последнюю запись, библиотека блокирует доступ к элементам пользовательского интерфейса для перемещения на последнюю и следующую запись.

В категории «Операции» имеется единственная функция, позволяющая программисту перемещаться по записям результирующего набора.

Функция

```
virtual BOOL CRecordView::OnMove(UINT nIDMoveCommand)
```

позволяет изменять указатель на текущую запись или, другими словами, перемещаться по записям результирующего набора и отображать его поля в элементах управления формы. Параметр nIDMoveCommand задает направление перемещения. Реализация этой функции по умолчанию обновляет текущую запись источника данных, если пользователь изменил ее в форме. Если результирующий набор не имеет записей, то вызов функции OnMove приводит к исключению. Поэтому перед ее использованием необходимо определить, имеются ли записи в результирующем наборе.

13. Доступ к данным в Visual C++

13.1. Интерфейсы доступа к данным

Количество доступных Windows-приложениям интерфейсов доступа к данным достаточно велико. Какую же из технологий – DAO, ODBC, RDO, UDA, OLE DB или ADO – выбрать для построения конкретного приложения?

Раньше в Microsoft ключевыми технологиями доступа к данным считались Data Access Objects (DAO) для настольных систем и Remote Data Objects (RDO), основанная на Open Database Connectivity (ODBC), для клиент-серверных баз данных. Но на смену им пришла единая модель Universal Data Access (UDA), поддерживающая данные любых типов.

Цель UDA – обеспечить высокопроизводительный доступ как к нереляционным, так и к реляционным источникам данных, предоставив удобный, независимый от инструментальных средств и языка интерфейс программирования. UDA базируется на объектах ADO (ActiveX Data Objects), которые предоставляют высокоуровневый интерфейс для работы с OLE DB – эффективной технологией Microsoft для доступа к базам данных на основе COM.

Хотя нет никаких ограничений в применении старых технологий для доступа к данным, при создании новых приложений лучше пользоваться UDA. Эта технология проста в обращении, характеризуется широким спектром возможностей и достаточной производительностью. Опытные разработчики программ на базе COM могут напрямую обращаться к интерфейсам OLE DB, получая выигрыш в скорости и эффективности.

Планируя перенос существующих DAO/ODBC-приложений на ADO, необходимо оценить, покроют ли преимущества от использования этой технологии издержки на ее внедрение, ведь код, написанный для DAO или RDO, напрямую не переносим на ADO. Тем не менее в подавляющем большинстве случаев решения, основанные на других моделях, можно с равным успехом реализовать на основе ADO.

13.2. Data Access Objects

Data Access Objects (DAO) – «родной» интерфейс программирования процессора базы данных Microsoft Jet, первоначально создавался для инструментальных сред разработки приложений Visual Basic и Visual Basic for Applications. DAO применяется Microsoft Jet для предоставления набора объектов доступа к данным, скрывающих стандартные объекты базы данных: таблицы, запросы и наборы записей (recordsets). Набор записей – это совокупность строк, возвращенных в ответ на запрос к базе данных.

Обычно DAO применяли для доступа к локальным источникам данных типа Microsoft Access, Microsoft FoxPro и Paradox, хотя сама технология вполне пригодна для доступа к удаленным источникам. На самом низком

уровне объекты DAO доступны через COM-интерфейсы, но чаще всего для доступа к ним применяются соответствующие MFC-классы DAO или же классы dbDAO. В C++ классы dbDAO предоставляют функциональность и синтаксис, сходные с Visual Basic.

13.3. Open Database Connectivity

Open Database Connectivity (ODBC) представляет собой API для доступа к клиент-серверным источникам данных, таким, как SQL Server или Oracle. Стандартный ODBC-интерфейс обеспечивает высокую степень универсальности приложения – один и тот же код применяется для взаимодействия с различными типами систем управления реляционными базами данных (СУРБД). Это позволяет разработчикам создавать и продавать клиент-серверные приложения, не ориентируясь на какую-то определенную СУРБД, а значит, не тратя силы и время на учет особенностей конкретных платформ серверов баз данных, с которыми работает приложение. ODBC-драйверы – все, что нужно такому приложению для взаимодействия с внешним источником данных. Эти драйверы в соответствии с открытым стандартом ODBC создают либо сами поставщики СУРБД, либо сторонние разработчики.

Возможности различных СУРБД иногда существенно отличаются. Кроме того, в ODBC-драйвере допускается реализация только некоторых из всех допустимых функций. По этой причине в ODBC определены три уровня соответствия (conformance) драйвера, позволяющие приложению узнавать о наборе функций, доступных в конкретном драйвере:

– базовое соответствие (core conformance) – минимум функций, обязательных для всех ODBC-драйверов;

– уровень соответствия 1 (level 1 conformance) – включает базовое соответствие и дополнительные функции, которые обычно применяются в СУРБД (например поддержка транзакций);

– уровень соответствия 2 (level 2 conformance) – включает уровень 1 плюс сложные функциональные возможности типа асинхронной работы ODBC.

Более подробную информацию об уровнях соответствия ODBC можно найти в справочной системе Visual C++.

На рис. 13.1 представлена архитекрура ODBC. Для установки и настройки ODBC-драйверов служит утилита ODBC Data Sources Control Panel. Она же предназначена и для регистрации нового имени источника (Data Source Name – DSN).

В качестве стандарта доступа к данным в ODBC применяется SQL. Когда приложению требуются данные из определенного источника, оно посылает SQL-запрос диспетчеру ODBC-драйверов, который в ответ загружает соответствующий ODBC драйвер. Тот, в свою очередь, преобразует поступивший из приложения SQL-запрос в понятную СУБД форму SQL и отсыла-

ет его серверу базы данных. Далее СУБД проводит выборку данных и через драйвер и диспетчер возвращает их приложению.

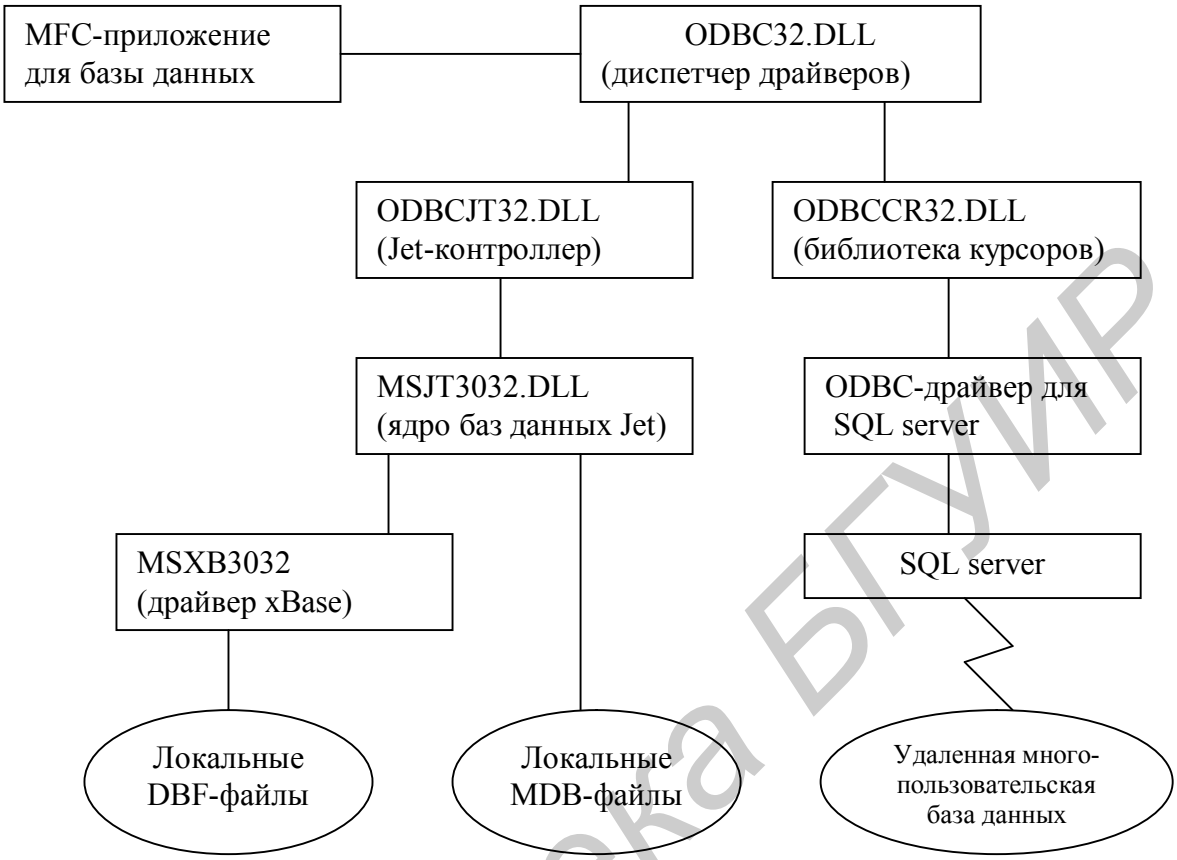

Рис. 13.1. Архитекрура ODBC

ODBC предоставляет библиотеку курсоров (cursor library) с курсорами прокрутки для драйверов, поддерживающих базовое соответствие ODBC. Они применяются для просмотра набора записей из базы данных.

13.4. Remote Data Objects

Remote Data Objects (RDO) является лишь объектно-ориентированной оболочкой ODBC API, а непосредственный доступ к данным выполняет ODBC-драйвер. Объектная модель RDO похожа на технологию DAO, но не требует дополнительной памяти для поддержки локальной базы данных. RDO предоставляет такие дополнительные функции, как *серверные курсоры* (server-side cursors), *отсоединенные наборы записей* (disconnected recordsets) и асинхронная обработка.

Как и DAO, для доступа к своим объектам RDO применяет COM-интерфейсы. Для этого служит Data Source Control – элемент управления на базе ActiveX, инкапсулирующий запросы к базам данных, и возвращаемые наборы записей. Он позволяет просматривать наборы записей с данными, ко-

торые содержатся в одном из Active-X-элементов для работы с данными – DBGrid или DBList.

13.5. OLE DB

OLE DB – набор COM-интерфейсов, предоставляющих приложению единообразный доступ к данным самых различных источников независимо от их местонахождения или типа. Открытая спецификация OLE DB основана на технологии ODBC, она предоставляет открытый стандарт доступа к данным любого типа. ODBC создавалась для взаимодействия с реляционными БД, а OLE DB разрабатывалась как для реляционных, так и для нереляционных источников, включая (но не ограничиваясь) БД на мейнфреймах, серверах и персональных компьютерах, а также хранилища файлов и сообщения электронной почты, электронные таблицы, инструментальные средства управления проектами и пользовательские объекты.

В соответствии с принципами построения OLE DB предусмотрено три типа компонентов: потребители данных (data consumers), служебные компоненты (service components) и поставщики данных (data providers). Потребители данных – это приложения или компоненты, которым нужны предоставляемые источником данные. Любое приложение, применяющее ADO, считается потребителем данных OLE DB. Служебные компоненты занимаются обработкой или транспортировкой данных, расширяя функциональнее возможности поставщиков данных. В качестве примера можно привести процессоры запросов (query processors), генерирующие или оптимизирующие запросы, и механизмы курсоров (cursor engines), принимающие данные из источников с последовательным доступом и отображающие их в удобной для просмотра форме. Поставщики данных, как ясно из их названия, представляют свои данные другим программам. Это могут быть как приложения, например SQL Server или Microsoft Exchange, так и системные компоненты – файловые системы или хранилища документов. Для доступа к данным поставщики предоставляют потребителям и служебным компонентам интерфейсы OLE DB. Предусмотрен и поставщик данных ODBC – он обеспечивает OLE DB-потребителям доступ ко всем существующим ODBC-источникам данных.

14. Потоки в Visual C++

В современных версиях Windows поддерживается два типа многозадачности. Первый тип основан на процессах. Процесс – это программа или задача, которая выполняется. В многозадачных системах такого типа две и более программы могут выполняться одновременно. Второй тип многозадачности основан на потоках. Поток – это часть выполняющегося процесса. В Win32 каждый процесс имеет по крайней мере один поток, но потоков процесса может быть и два и больше.

В потоковой многозадачности несколько частей одной и той же программы могут выполняться одновременно. Это дает возможность писать чрезвычайно эффективные программы путем разделения их на отдельные исполняемые блоки и управления ходом выполнения всей программы в целом. Для многозадачности такого типа в MFC предусмотрены специальные средства поддержки.

С введением потоковой многозадачности возникла необходимость в специальном механизме, называемом синхронизацией. Синхронизация позволяет контролировать выполнение потоков (и процессов) строго определенным образом. Библиотека классов MFC полностью поддерживает средства многозадачности.

14.1. Интерфейсные и рабочие потоки MFC

Все процессы имеют по крайней мере один поток выполнения. Он называется главным, первичным потоком. Но в пределах одного и того же процесса можно создавать несколько потоков. Когда они создаются, родительский процесс начинает выполняться не последовательно, а параллельно.

В MFC определены два типа потоков: интерфейсные и рабочие. Интерфейсный поток способен принимать и обрабатывать сообщения. Говоря языком MFC, интерфейсные потоки содержат канал сообщений. Главный поток MFC-программы (начинающийся при объявлении объекта класса CWinApp) является интерфейсным потоком. Рабочие потоки не принимают и не обрабатывают сообщения. Они обеспечивают дополнительные пути выполнения задачи внутри интерфейсного потока.

В MFC потоковая многозадачность реализуется с помощью класса CWinThread. Заметим, что производным от него является класс CWinApp, формирующий поток приложения. При использовании классов, отвечающих за работу в многозадачном режиме, в программу следует включать стандартный библиотечный файл afxmt.h.

При создании многопоточных программ наиболее часто используются именно рабочие потоки – необходимость в нескольких каналах сообщений возникает достаточно редко, однако во многих приложениях используются вспомогательные потоки, позволяющие вести фоновую обработку данных.

14.1.1. Создание рабочего потока

Для создания рабочего потока предназначена функция AfxBeginThread библиотеки MFC:

```
CWinThread* AfxBeginThread( AFX_THREADPROC pfnThreadProc,
     LPVOID pParam, int nPriority = THREAD_PRIORITY_NORMAL,
     UINT nStackSize = 0, DWORD dwCreateFlags = 0,
     LPSECURITY_ATTRIBUTES lpSecurityAttrs = NULL );
```

Каждый поток внутри родительского процесса начинает свое выполнение с вызова специальной функции, называемой потоковой функцией. Выполнение потока продолжается до тех пор, пока не завершится его потоковая функция. Адрес данной функции (т. е. входная точка в поток) передается в параметре pfnThreadProc. Все потоковые функции должны иметь следующий прототип:

```
UINT pfnThreadProc (LPVOID pParam);
```

Значение параметра pParam функции AfxBeginThread есть 32-разрядное число, которое может использоваться для любых целей.

Начальный приоритет потока указывается в параметре nPriority. Если этот параметр равен 0, то используются установки приоритета текущего (родительского) потока. Каждый поток имеет свой собственный стек. Размер стека указывается в параметре nStackSize. Если этот параметр равен нулю (общепринятый подход), то создаваемому потоку будет выделен стек такого же размера, что и у родительского потока, а при необходимости размер стека может быть увеличен.

Параметр dwCreateFlags определяет состояние выполнения потока. Если данный параметр равен нулю, поток начинает выполняться немедленно. Если значение этого параметра равно CREATE_SUSPEND, то поток создается временно приостановленным, т. е. ожидающим запуска. Чтобы запустить такой поток, нужно вызвать функцию CWinThread::ResumeThread.

Параметр lpSecurityAttrs является указателем на набор атрибутов прав доступа, относящийся к данному потоку. Если этот параметр равен NULL, то набор атрибутов будет унаследован от родительского окна.

При успешном завершении функция AfxBeginThread возвращает указатель на объект потока, в противном случае возвращает ноль. Данный указатель необходимо сохранять, если впоследствии предполагается обращение из родительского потока к созданному потоку (например для изменения приоритета или для временного приостановления потока).

В программе может быть столько потоков, сколько необходимо. При работе с несколькими потоками для каждого из них должна быть определена своя потоковая функция и каждый из них должен начинаться отдельно. Все потоки процесса затем функционируют одновременно.

Рассмотрим пример создания двух потоков для однодокументного приложения Example при обработке сообщения о выборе пользователем пункта меню Start Thread меню Thread. В качестве родительского потока выступает главный

поток приложения. Поток 1 после запуска осуществляет 100-кратный вывод некоторой строки в окно приложения с задержкой 650 миллисекунд, поток 2 каждые две секунды 50 раз выдает звуковой сигнал и сообщение.

Для создания приложения Example выполните следующие действия.

1. Запустите AppWizard и укажите ему на необходимость создания нового проекта класса MFC AppWizard(exe) с именем Example.

2. Задайте для нового проекта параметры настройки AppWizard: шаг 1– SDI, остальные по умолчанию.

3. Используя редактор ресурсов, добавьте в меню приложения IDR_MAINFRAME новое меню Thread. Поместите в него команду с названием Start Thread и идентификатором ID_STARTTHREAD.

4. С помощью ClassWizard свяжите команду ID_STARTTHREAD с функцией обработки сообщения OnStartthread(). Перед добавлением этой функции убедитесь, что в поле Class Name выбрано значение CExampleView.

5. Щелкните на кнопке Edit Code и введите приведенные ниже операторы в новую функцию OnStartthread().

```
AfxBeginThread(MyThread1,this);
AfxBeginThread(MyThread2,this);
```

В этом фрагменте текста программы последовательно вызываются функции MyThread1() и MyThread2(), каждая из них будет работать в своем собственном потоке. Далее в файл ExampleView.cpp добавьте функции MyThread1() и MyThread2(), текст которых представлен ниже. Поместите перед функцией OnStartthread() объявления функций MyThread1() и MyThread2(). Обратите внимание, что эти функции являются глобальными функциями, а не методами класса CExampleView, несмотря на то что они находятся в файле, в котором реализован этот класс.

Окончательный фрагмент кода в файле ExampleView.cpp представлен в листинге 14.1.

Листинг 14.1.

```
UINT MyThread1(LPVOID pParam); // объявление функции потока 1
UINT MyThread2(LPVOID pParam); // объявление функции потока 2
…
void CExampleView::OnStartthread() //обраб. сообщения от меню
{//Создать два новых потока. Функция потока 1 имеет имя
//MyThread1, Функция потока 2 имеет имя MyThread2.
// в качестве параметра функциям потоков передается указатель
// на текущее окно просмотра для вывода в него изображения
     AfxBeginThread(MyThread1,this);
     AfxBeginThread(MyThread2,this);
}
// определение функции потока 1
UINT MyThread1(LPVOID pParam)
{// через параметр передается указатель на окно просмотра
     CExampleView *ptrView=(CExampleView *)pParam;
     for(int i=0; i<100; i++)
```

```
     {CDC *dc=ptrView->GetDC();// получить контекст отображения
     Sleep(650); // Задержка на 650 миллисекунд
     CRect r;
     ptrView->GetClientRect(&r); //получить клиентскую область окна
dc->TextOut(rand()%r.Width(),rand()%r.Height(),"*",1); // вывод
     }
     return 0;
}
// определение функции потока 2
UINT MyThread2(LPVOID pParam)
{for(int i=0; i<50; i++)
{    Sleep(2000); // Задержка на 2000 миллисекунд
     AfxMessageBox("MyThread2"); // Вывод сообщения
     MessageBeep(0); } // Подача звукового сигнала

return 0;
}
```

Откомпилируйте и запустите приложение. Установите при компиляции на странице свойств проекта Example опцию многопоточного приложения (рис. 14.1).

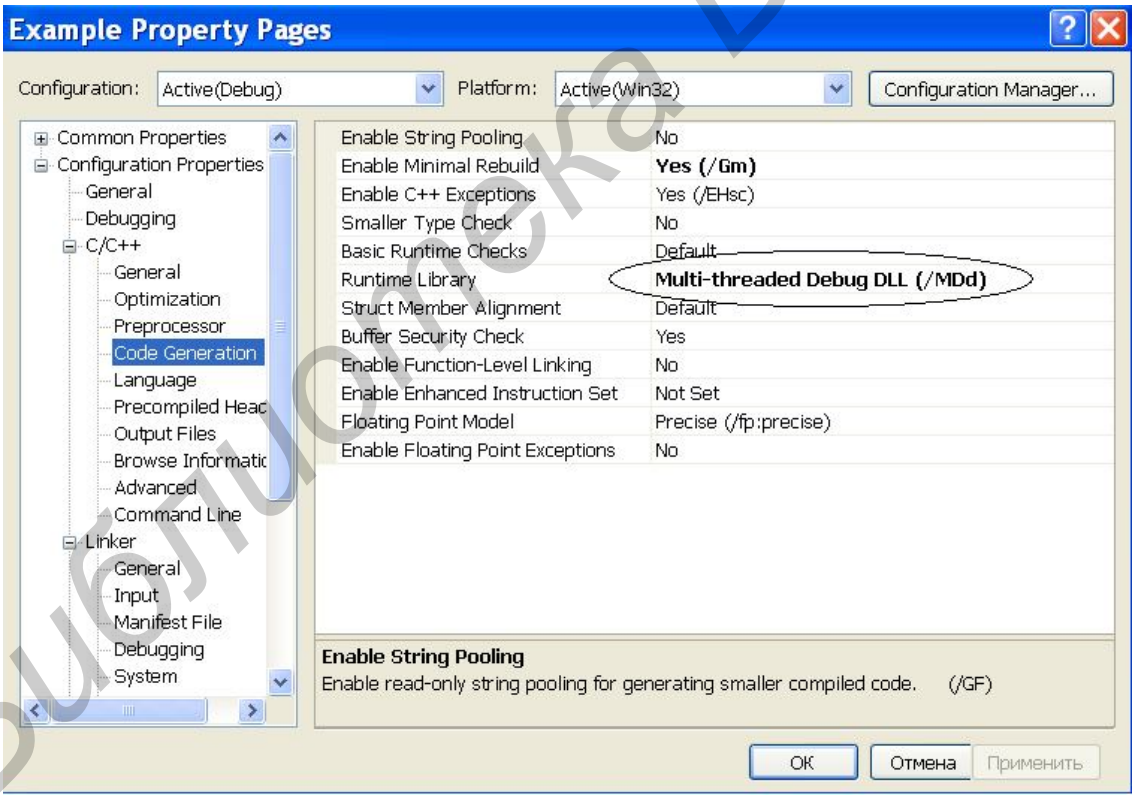

Рис. 14.1. Настройка проекта на многопоточность

Иногда бывает необходимо приостановить поток на заданное количество миллисекунд. Это можно сделать, вызвав API-функцию Sleep.

Вообще говоря, поток выполняется до завершения своей потоковой функции. Поток может также «завершить сам себя» с помощью функции AfxEndThread библиотеки MFC. Параметр этого метода содержит статус за-

вершения потока. Как правило, лучше давать потоку возможность нормально завершиться одновременно с потоковой функцией.

14.1.2. Остановка и возобновление выполнения потоков

Остановить выполнение потока можно с помощью метода SuspendThread класса CWinThread. В остановленном состоянии поток не выполняется. Продолжить выполнение потока можно с помощью метода ResumeThread класса CWinThread.

Каждый поток имеет связанный с ним счетчик остановок. Если этот счетчик равен нулю, значит, поток выполняется нормально. При ненулевом значении счетчика поток находится в остановленном состоянии. С каждым вызовом метода SuspendThread значение счетчика остановок увеличивается на единицу. И, наоборот, с каждым вызовом функции ResumeThread значение счетчика остановок уменьшается на единицу. Остановленный поток может продолжить выполнение только после того, как значение счетчика достигнет нуля.

14.2. Управление приоритетами потоков

С каждым потоком связана определенная установка приоритета. Эта установка представляет собой комбинацию двух значений: значения общего класса приоритета процесса и значения приоритета самого потока относительно данного класса.

Приоритет потока показывает, сколько времени работы процессора требуется потоку. Для потоков с низким приоритетом требуется мало времени, а для потоков с высоким приоритетом – много времени.

Получить класс приоритета процесса можно с помощью функции GetPriorityClass, а установить класс приоритета можно с помощью функции SetPriorityClass. Обе эти функции являются API-функциями и не входят в класс CWinThread.

В табл. 14.1 показаны константы, соответствующие классам приоритетов в порядке убывания (по умолчанию программе присваивается приоритет NORMAL_PRIORITY_CLASS; причин менять его, как правило, нет).

Изменение приоритета процесса может негативно сказаться на производительности всей системы. Так, например, увеличение класса приоритета программы до REALTIME_PRIORITY_CLASS приведет к захвату программой всех ресурсов процессора.

Приоритет потока процесса (независимо от класса приоритета) говорит о том, сколько времени процессора занимает отдельный поток в пределах своего процесса. При создании потока ему присваивается нормальный приоритет THREAD_PRIORITY_NORMAL. Но это значение можно изменить, причем даже во время выполнения потока.

Таблица 14.1

Классы приоритета процесса

Приоритет процесса	Значение
REALTIME_PRIORITY_CLASS	Наиболее высокий приоритет
HIGH_PRIORITY_CLASS	Выполнение критических по времени задач
NORMAL_PRIORITY_CLASS	Без специальных требований
IDLE_PRIORITY_CLASS	Запускается только, когда система не занята

Приоритеты потоков контролируются методами класса CWinThread. Определить значение приоритета можно с помощью метода GetThreadPriority, а изменить его – с помощью метода SetThreadPriority. В табл. 14.2 приведены константы, соответствующие установкам приоритетов потоков в порядке убывания.

Таблица 14.2

Классы приоритета потока

Константа	Назначение
THREAD_PRIORITY_TIME_CRITICAL	Устанавливает базовый приоритет равным 15. Для процесса realtime_priopity_class устанавливает приоритет равным 30
THREAD_PRIORITY_HIGHEST	Устанавливает приоритет на два пункта выше нормального
THREAD_PRIORITY_ABOVE_NORMAL	Устанавливает приоритет на один пункт выше нормального
THREAD_PRIORITY_NORMAL	Устанавливает нормальный приоритет
THREAD_PRIORITY_BELOW_NORMAL	Устанавливает приоритет на один пункт ниже нормального
THREAD_PRIORITY_LOWEST	Устанавливает приоритет на два пункта ниже нормального
THREAD_PRIORITY_IDLE	Устанавливает базовый приоритет равным 1. Для процесса realtime_priority_class устанавливает приоритет равным 16

Благодаря различным сочетаниям значений приоритета процесса и приоритета потока, в Win32 поддерживается 31 различная установка приоритета.

14.3. Синхронизация потоков

Использование в программе нескольких потоков одновременно может привести к возникновению ряда специфических проблем. Например, как предотвратить одновременный доступ двух потоков к одним и тем же дан-

ным? Что произойдет, если в тот момент, когда один поток еще не завершил процедуру обновления некоторых данных, другой поток предпринимает попытку эти данные считать? Почти наверняка данные, считанные вторым потоком, окажутся некорректными, поскольку лишь некоторая их часть была на данный момент обновлена.

Обеспечение корректной совместной работы потоков называется синхронизацией потоков. Рассмотрим средства синхронизации потоков.

14.3.1. Объекты синхронизации и классы MFC

Интерфейс Win32 поддерживает четыре типа объектов синхронизации – все они, так или иначе, основаны на понятии семафора.

Первым типом объектов является классический (стандартный) семафор. Он позволяет ограниченному числу процессов и потоков обращаться к одному ресурсу. При этом доступ к ресурсу либо полностью ограничен (один и только один поток или процесс может обратиться к ресурсу в определенный период времени), либо одновременный доступ получает лишь малое количество потоков и процессов. Семафоры реализуются с помощью счетчика, значение которого уменьшается, когда задаче выделяется семафор, и увеличивается, когда задача освобождает семафор.

Вторым типом объектов синхронизации является исключающий (mutex) семафор. Он позволяет в любой момент времени обратиться к ресурсу только одному процессу или потоку.

Третьим типом объектов синхронизации является событие, или объект события (event object). Он используется для блокирования доступа к ресурсу до тех пор, пока какой-нибудь другой процесс или поток не заявит о том, что данный ресурс может быть использован.

При помощи объекта синхронизации четвертого типа можно запрещать выполнения определенных участков кода программы несколькими потоками одновременно. Для этого данные участки должны быть объявлены как критическая секция (critical section). Когда в эту секцию входит один поток, другим потокам запрещается входить в нее до тех пор, пока первый поток не выйдет из данной секции. Критические секции, в отличие от других типов объектов синхронизации, применяются только для синхронизации потоков внутри одного процесса. Другие же типы объектов могут быть использованы для синхронизации потоков внутри процесса или для синхронизации процессов.

Все классы MFC, реализующие механизм синхронизации, можно разделить на две категории:

– классы для синхронизации работы потоков;
– классы для контроля доступа к объекту синхронизации.

Для синхронизации работы потоков используются следующие классы:

– CCriticalSection – реализует критическую секцию;

– CEvent – реализует объект события;
– CMutex – реализует исключающий семафор;
– CSemaphore – реализует классический семафор.

Для контроля доступа используются следующие классы: CSingleLock и CMultiLock. Они контролируют доступ к объекту синхронизации и содержат методы, используемые для предоставления и освобождения таких объектов. Класс CSingleLock управляет доступом к одному объекту синхронизации, а класс CMultiLock – к нескольким объектам.

Когда какой-либо объект синхронизации создан, доступ к нему можно контролировать с помощью класса CSingleLock. Для этого необходимо сначала создать объект типа CSingleLock с помощью конструктора:

```
CSingleLock(CSyncObject* pObject, BOOL bInitialLock=FALSE);
```

Через первый параметр передается указатель на объект синхронизации, например семафор. Значение второго параметра определяет, должен ли конструктор попытаться получить доступ к данному объекту. Если этот параметр не равен нулю, то доступ будет получен, в противном случае попыток получить доступ не будет. Если доступ получен, то поток, создавший объект класса CSingleLock, будет остановлен до освобождения соответствующего объекта синхронизации методом Unlock класса CSingleLock.

Когда объект типа CSingleLock создан, доступ к объекту, на который указывал параметр pObject, может контролироваться с помощью двух функций: Lock и Unlock класса CSingleLock.

Метод Lock() предназначен для получения доступа к объекту синхронизации. Если объект синхронизации в данный момент не захвачен другим потоком, функция Lock() передаст этот объект во владение данному потоку. Теперь поток может получить доступ к защищенным данным. Завершив обработку данных, поток должен вызвать метод Unlock(), который освобождает объект синхронизации, давая возможность другим потокам использовать ресурс.

При работе с классом CSingleLock общая процедура управления доступом к ресурсу такова:

– создать объект синхронизации (например, семафор), который будет использоваться для управления доступом к ресурсу;
– с помощью созданного объекта синхронизации создать объект типа CSingleLock;
– для получения доступа к ресурсу вызвать метод Lock();
– выполнить обращение к ресурсу;
– вызвать метод Unlock(), чтобы освободить ресурс.

Важно уметь определять тот класс, который нужен для работы. Если приложение должно ждать некоторого события перед получением доступа, то нам нужен CEvent. Если к объекту будут иметь доступ несколько потоков

из одного приложения и нужны ограничения по количеству потоков, то нам нужен CSemaphore. Если к объекту будет иметь доступ только один поток и из одного приложения, то CCriticalSection. Если к объекту будет иметь доступ только один поток, но из разных приложений, то CMutex.

Рассмотрим, как создавать и использовать объекты синхронизации.

14.3.2. Работа с семафорами

Рассмотрим, как обеспечить синхронизацию потоков на основе семафоров. Прежде всего необходимо создать семафор путем объявления объекта типа CSemaphore. Конструктор этого класса имеет следующий вид:

```
CSemaphore(LONG lInitialCount=1,LONG lMaxCount=1,
          LPCTSTR pstrName=NULL,
          LPSECURITY_ATTRIBUTES lpsaAttributes=NULL);
```

Семафоры имеют счетчик, указывающий количество задач, которым в настоящее время предоставлен доступ к ресурсу. Если значение счетчика равно нулю, то последующий доступ к ресурсу запрещается до тех пор, пока одна из задач не освободит семафор. Начальное значение счетчика семафора указывается в первом параметре конструктора. Обычно начальное значение задается равным единице, чтобы хотя бы один поток мог получить семафор. Допустимое число потоков, которым будет разрешен одновременный доступ, указывается во втором параметре. Если это значение равно единице, то семафор будет исключающим.

Третий параметр конструктора указывает на строку, содержащую имя объекта семафора. Поименованные семафоры становятся системными объектами и могут использоваться другими процессами. Когда два процесса вызывают семафоры с одинаковыми именами, обоим процессам будет предоставлен один и тот же семафор – это позволяет синхронизировать процессы. Вместо имени строки можно указать NULL – в этом случае семафор будет локализован внутри одного процесса. Последний параметр конструктора является указателем на набор атрибутов прав доступа, связанный с семафором. Если этот параметр равен NULL, то семафор наследует данный набор у вызвавшего его потока.

Модифицируйте приложение Example, добавив в него функции, использующие семафор. Для этого добавьте в меню Thread пункт Semaphore. Функция OnSemaphore(), реализующая этот пункт, создает три потока, которые используют один и тот же ресурс. Одновременно доступ к ресурсу могут получить только два потока. Третий должен ждать, когда ресурс освободится.

Создавая семафор, вы передаете ему начальное и максимальное значения счетчика, как показано ниже:

```
CSemaphore Semaphore(2, 2);
```

Поскольку в этом примере семафоры будут использоваться для создания потокового класса, логично будет объявить указатель на объект класса CSemaphore в качестве переменной-члена потокового класса, а затем динамически создать объект класса CSemaphore в конструкторе потокового класса, как показано ниже:

```
semaphore = new CSemaphore(2, 2);
```

Теперь, когда объект семафора создан, можно начинать отсчет количества обращений к ресурсу. Для реализации процесса подсчета прежде всего необходимо создать экземпляр класса CSingleLock, передав ему указатель на семафор, который вы хотите использовать:

```
CSingleLock singleLock(semaphore);
```

Затем для уменьшения значения счетчика семафора вызывается метод Lock() класса CSingleLock:

```
singleLock.Lock();
```

На данный момент объект семафора выполнил уменьшение значения своего внутреннего счетчика. Это новое значение сохраняется до тех пор, пока объект семафора не будет освобожден посредством вызова его метода Unlock():

```
singleLock.Unlock();
```

Если сразу после освобождения семафора происходит выход объекта класса CSingleLock из области видимости (завершение функции, в которой он объявлен), метод Unlock() для объекта singleLock можно не вызывать. Деструктор объекта singleLock, вызванный при завершении работы функции, выполнит Unlock() автоматически. Доступ к разделяемому ресурсу осуществим в классе CSomeResource. Класс имеет единственную переменную-член, являющуюся указателем на объект класса CSemaphore. Кроме того, в классе определены конструктор и деструктор, а также метод UseResource(), в котором непосредственно используется семафор. Файл заголовка и файл реализации класса SomeResource представлены в листинге 14.2.

Листинг 14.2.

```
//Файл заголовка SomeResource.h
#include "afxmt.h"
class CSomeResource
```

```
{
private:
     CSemaphore* semaphore;
public:
     CSomeResource();
     ~CSomeResource();
void UseResource();
};
//Файл реализации SomeResource.cpp:
#include "stdafx.h"
#include "SomeResource.h"
CSomeResource::CSomeResource()
{
     semaphore = new CSemaphore(2,2);
}
CSomeResource::~CSomeResource()
{
     delete semaphore;
}
void CSomeResource::UseResource()
{
     CSingleLock singleLock(semaphore);
     singleLock.Lock();
     Sleep(5000);
}
```

В тексте файла, реализующего класс CSomeResource, можно видеть, что объект класса CSemaphore динамически создается в конструкторе класса CSomeResource и уничтожается в его деструкторе. Метод UseResource() эмулирует доступ к ресурсу. Он захватывает семафор, затем ожидает 5 секунд и вновь его освобождает. Модифицируйте приложение Example следующим образом. Добавьте в меню Thread пункт Semaphore и функцию OnSemaphore() в класс CExampleView. Добавьте в проект два новых пустых файла SomeResource.h и SomeResource.cpp. Добавьте в эти файлы тексты программ, приведенные выше. Добавьте в файл ExampleView.cpp после директивы

```
#include "ExampleView.h"
```

директиву

```
#include "SomeResource.h"
```

Включите в начало файла сразу же после директивы #endif строку

```
CSomeResource someResource;
```

Добавьте в файл ExampleView.cpp перед функцией CExampleView::OnSemaphore() три следующие функции:

```
UINT ThreadProc1(LPVOID pParam)
{
     someResource.UseResource();
     AfxMessageBox("Thread1 had access.");
     return 0;
}

UINT ThreadProc2(LPVOID pParam)
{
     someResource.UseResource();
     AfxMessageBox("Thread2 had access.");
     return 0;
}

UINT ThreadProc3(LPVOID pParam)
{
     someResource.UseResource();
     AfxMessageBox("Thread3 had access.");
     return 0;
}
```

Добавьте в функцию CExampleView::OnSemaphore() следующие строки:

```
AfxBeginThread(ThreadProc1, this);
AfxBeginThread(ThreadProc2, this);
AfxBeginThread(ThreadProc3, this);
```

Теперь откомпилируйте новую версию приложения Example и запустите ее на выполнение. В раскрывшемся главном окне приложения выберите команду Threads->Semaphore. Приблизительно через 5 секунд появятся два окна сообщений, информирующих о том, что первый и второй потоки получили доступ к защищенному ресурсу. Еще через 5 секунд появится третье окно сообщений, в котором говорится о том, что третий поток также получил доступ к ресурсу. Третьему потоку потребовалось на 5 секунд больше по той причине, что первые два потока первыми захватили контроль над ресурсом. Семафор в этой программе организован таким образом, что разрешает доступ к ресурсу только двум потокам одновременно. Таким образом, третий поток вынужден был ожидать, пока первый или второй поток освободит защищенный ресурс.

15. Создание и использование динамически связываемых библиотек

С самого появления операционная система Windows использовала библиотеки динамической компоновки DLL (Dynamic Link Library), в которых содержались реализации наиболее часто применяемых функций. Практически невозможно создать приложение Windows, в котором не использовались бы библиотеки DLL. В DLL содержатся все функции Win32 API и большое количество других функций операционных систем Win32. Вообще говоря, DLL – это просто наборы функций, собранные в библиотеки. Однако в отличие от статических библиотек (файлов.lib), библиотеки DLL не присоединены непосредственно к выполняемым файлам с помощью редактора связей.

В выполняемый файл занесена только информация об их местонахождении. В момент выполнения программы загружается вся библиотека целиком. Благодаря этому разные процессы могут пользоваться совместно одними и теми же библиотеками, находящимися в памяти. Такой подход позволяет сократить объем памяти, необходимый для нескольких приложений, использующих много общих библиотек, а также контролировать размеры EXE-файлов.

Однако если библиотека используется только одним приложением, лучше сделать ее обычной, статической. Конечно, если входящие в ее состав функции будут использоваться только в одной программе, можно просто вставить в нее соответствующий файл с исходным текстом.

Чаще всего проект подключается к DLL статически, или неявно, на этапе компоновки. Загрузкой DLL при выполнении программы управляет операционная система. Однако DLL можно загрузить и явно, или динамически, в ходе работы приложения.

15.1. Статическое подключение DLL

При статическом подключении DLL имя соответствующего ей lib-файла определяется среди прочих параметров редактора связей в командной строке или в окне Linker окна свойств проекта.

Однако lib-файл, используемый при неявном подключении DLL, – это не обычная статическая библиотека. Такие lib-файлы называются библиотеками импортирования (import libraries). В них содержится не сам код библиотеки, а только ссылки на все функции, экспортируемые из файла DLL, в котором все и хранится. В результате библиотеки импортирования, как правило, имеют меньший размер, чем DLL-файлы.

При загрузке неявно подключаемой DLL приложение пытается найти все файлы DLL, неявно подключенные к приложению, и поместить их в область оперативной памяти, занимаемую данным процессом. Поиск файлов DLL операционной системой осуществляется в следующей последовательности:

– каталог, в котором находится EXE-файл;
– текущий каталог процесса;
– системный каталог Windows\System.

Если библиотека DLL не обнаружена, приложение выводит диалоговое окно с сообщением о ее отсутствии и путях, по которым осуществлялся поиск. Затем процесс отключается. Если нужная библиотека найдена, она помещается в оперативную память процесса, где и остается до его окончания. Теперь приложение может обращаться к функциям, содержащимся в DLL.

15.2. Динамическая загрузка и выгрузка DLL

Вместо того чтобы Windows выполняла динамическое связывание с DLL при первой загрузке приложения в оперативную память, можно связать программу с модулем библиотеки во время выполнения программы (при таком способе в процессе создания приложения не нужно использовать библиотеку импорта).

В частности, можно определить, какая из библиотек DLL доступна пользователю, или разрешить пользователю выбрать, какая из библиотек будет загружаться. Таким образом, можно использовать разные DLL, в которых реализованы одни и те же функции, выполняющие различные действия. Например, приложение, предназначенное для независимой передачи данных, сможет в ходе выполнения принять решение, загружать ли DLL для протокола TCP/IP или для другого протокола.

Первое, что необходимо сделать при динамической загрузке DLL, – это поместить модуль библиотеки в память процесса. Данная операция выполняется с помощью функции ::LoadLibrary, имеющей единственный аргумент – имя загружаемого модуля.

Стандартным расширением файла библиотеки Windows считает .dll, если не указать другое расширение. Если в имени файла указан и путь, то только он будет использоваться для поиска файла. В противном случае Windows будет искать файл по той же схеме, что и в случае неявно подключенных DLL. Когда Windows обнаружит файл, его полный путь будет сравнен с путем библиотек DLL, уже загруженных данным процессом. Если обнаружится тождество, вместо загрузки копии приложения возвращается дескриптор уже подключенной библиотеки.

Если файл обнаружен и библиотека успешно загрузилась, функция ::LoadLibrary возвращает ее дескриптор, который используется для доступа к функциям библиотеки. После завершения работы с библиотекой динамической компоновки ее можно выгрузить из памяти процесса с помощью функции ::FreeLibrary.

Файл библиотеки также несколько отличается от обычных файлов на языке C++ для Windows. В нем вместо функции WinMain имеется функция DllMain.

После трансляции и компоновки файлов библиотеки появляются два файла (например для библиотеки MyDLL) – MyDLL.dll (сама динамически подключаемая библиотека) и MyDLL.lib (ее библиотека импорта).

При неявном подключении DLL следует в исходный текст помимо вызова функции MyFunction из DLL-библиотеки включить и заголовочный файл этой библиотеки MyDLL.h.

Также необходимо на этапе компоновки приложения подключить к нему библиотеку импорта MyDLL.lib (процесс неявного подключения DLL к исполняемому модулю). Сам код функции MyFunction не включается в файл приложения MyApp.exe. Вместо этого там просто имеется ссылка на файл MyDLL.dll и ссылка на функцию MyFunction, которая находится в этом файле. Файл MyApp.exe требует запуска файла MyDLL.dll.

15.3. Создание DLL

Рассмотрим способы их создания DLL. При разработке приложений функции, к которым обращается несколько процессов, желательно размещать в DLL. Это позволяет более рационально использовать память в Windows.

Проще всего создать новый проект DLL с помощью мастера AppWizard, который автоматически выполняет многие операции. Для простых DLL необходимо выбрать тип проекта Win32 Dynamic-Link Library. Новому проекту будут присвоены все необходимые параметры для создания библиотеки DLL. Файлы исходных текстов придется добавлять к проекту вручную.

Если же планируется в полной мере использовать функциональные возможности MFC, такие как документы и представления, лучше выбрать тип проекта MFC AppWizard (dll). В этом случае помимо присвоения проекту параметров для подключения динамических библиотек мастер проделает некоторую дополнительную работу. В проект будут добавлены необходимые ссылки на библиотеки MFC и файлы исходных текстов, содержащих описание и реализацию в библиотеке DLL объекта класса приложения, производного от CWinApp.

Большинство библиотек DLL – просто коллекции практически независимых друг от друга функций, экспортируемых в приложения и используемых в них. Кроме функций, предназначенных для экспортирования, в каждой библиотеке DLL есть функция DllMain. Эта функция предназначена для инициализации и очистки DLL. Функция DllMain вызывается в нескольких случаях: при первой загрузке библиотеки DLL процессом, каждый раз при создании процессом нового потока, при уничтожении потока (кроме первого), по окончании работы процесса с DLL. Если не написать собственной функции DllMain, компилятор подключит стандартную версию, которая просто возвращает TRUE.

15.4. Экспортирование функций из DLL

Чтобы приложение могло обращаться к функциям динамической библиотеки, каждая из них должна занимать строку в таблице экспортируемых функций DLL. Есть два способа занести функцию в эту таблицу на этапе компиляции.

Можно экспортировать функцию из DLL, поставив в начале ее описания модификатор __declspec (dllexport). Этот метод применяется не так часто, как второй метод, работающий с файлами определения модуля (.def), и позволяет лучше управлять процессом экспортирования. def-Файл содержит имя и описание библиотеки, а также список экспортируемых функций:

```
MyDLL.def
     LIBRARY        "MyDLL"
     DESCRIPTION    'MyDLL – пример DLL-библиотеки'
     EXPORTS
          MyFunction     @1
```

В строке экспорта функции можно указать ее порядковый номер, поставив перед ним символ @. Этот номер будет затем использоваться при обращении к функции GetProcAddress. На самом деле компилятор присваивает порядковые номера всем экспортируемым объектам. Однако способ, которым он это делает, отчасти непредсказуем, если не присвоить эти номера явно. В строке экспорта можно использовать параметр NONAME. Он запрещает компилятору включать имя функции в таблицу экспортирования DLL:

```
     MyFunction     @1 NONAME
```

Иногда это позволяет сэкономить много места в файле DLL. Приложения, использующие библиотеку импортирования для неявного подключения DLL, не заметят разницы, поскольку при неявном подключении порядковые номера используются автоматически. Приложениям, загружающим библиотеки DLL динамически, потребуется передавать в GetProcAddress порядковый номер, а не имя функции.

15.5. Использование динамически связываемых библиотек

Рассмотрим пример создания библиотеки DLL, содержащей системную функцию, возвращающую размер свободного дискового пространства. Для определения экспортируемой функции используется следующий синтаксис.

Для переменных:

```
__declspec(dllexport) <тип данных> <идентификатор переменной>;
```

Для функций:

```
__declspec(dllexport) <возвращаемый тип> <имя функции> ([списокАргументов]);
```

Импортирование функций организуется практически так же – просто замените ключевое слово, например `__declspec(dllexport)`, словом

__declspec(dllimport). Используя реальные функцию и переменную для демонстрации синтаксиса, получим следующее:

```
__declspec(dllimport) int referenceCount;
__declspec(dllimport) void DiskFree(lpStr Drivepath );
```

Ключевому слову __declspec предшествуют два знака подчеркивания.

Чтобы упростить описание DLL-модулей, Microsoft использует файл заголовка и макросы препроцессора. Эта методика требует использовать уникальную лексему препроцессора (проще всего для этого использовать имя файла заголовка) и написать макрос, который будет замещать эту лексему корректными операторами импорта и экспорта. Предположим, что имеется файл заголовка с именем DISKFREE.H, тогда макрос препроцессора в нем может выглядеть так, как показано в листинге 15.1.

Листинг 15.1.

```
//DISKFREE.H содержит простую функцию, возвращающую размер
// свободного дискового пространства.
#ifndef __DISKFREE_H
#define __DISKFREE_H
#ifndef __DISKFREE__
#define DISKFREELIB __declspec(dllimport)
#else
#define DISKFREELIB __declspec(dllexport)
#endif
//Макрос используется для выбора описания импорта или экспорта.
DISKFREELIB unsigned long DiskFree( unsigned int drive );
//Например, 0 = A:, 1=B:, 2 = C:
#endif
```

Подключив файл заголовка, вы даете возможность препроцессору определить, импортируется или экспортируется функция DiskFree. Теперь вы можете предоставить такой файл заголовка как разработчику, так и пользователю модуля DLL, что решает много проблем при сопровождении программ.

15.5.1. Создание модуля DLL DiskFree

Утилита DiskFree позволяет легко определять размер свободного пространства на указанном диске. Ее работа основана на использовании функции GetDiskFreeSpace(), находящейся в модуле Kernel32.Dll.

Для того чтобы создать не использующий MFC модуль DLL, выберите тип проекта Win32Dll. Назовите проект DiskFree, выберите опции DLL и Empty project. В результате будет создан проект, пока еще не имеющий собственных файлов.

Добавьте в проект файл заголовка DiskFree.h и файл текста программы DiskFree.cpp и поместите в них текст, представленный в листинге 15.2. Теперь можно оттранслировать модуль DLL. Если при компиляции модуля DiskFree использовать установки по умолчанию, то модуль DLLMain будет

применяться неявно (его добавляет компилятор) и будет использоваться режим неявной загрузки, причем Windows будет автоматически управлять загрузкой и выгрузкой этой библиотеки.

Листинг 15.2.

```
// файл заголовка DiskFree.h
#ifndef __DISKFREE_H
#define __DISKFREE_H
#ifndef __DISKFREE__
#define __DISKFREELIB__ __declspec(dllimport)
#else
#define __DISKFREELIB__ __declspec(dllexport)
#endif
//Возвращает размер свободного дискового пространства на диске,
//заданном его номером (например, 0=A:, 1=B:, 2=C:)
__DISKFREELIB__ unsigned long DiskFree( unsigned int drive );
#endif
// файл текста программы DiskFree.cpp
#include <afx.h>
#include <winbase.h>                //Содержит объявление функции
//GetDiskFreeSpace(),находящейся в модуле kernel32.dll.
#define  __DISKFREE__  //Определяет лексему перед подключением
//библиотеки.
#include "diskfree.h"
// Возвращает размер свободного дискового пространства на диске,
// заданном его номером (например, 0 = A:, 1= B:, 2 = C:).
__DISKFREELIB__ unsigned long DiskFree( unsigned int drive )
{
unsigned long bytesPerSector, sectorsPerCluster,
freeClusters, totalClusters;
char DrivePath[4] = { char (drive + 65), ':', '\\', '\0' };
if( GetDiskFreeSpace( DrivePath, &sectorsPerCluster,
&bytesPerSector, &freeClusters, &totalClusters )) {
return sectorsPerCluster * bytesPerSector * freeClusters; )
}
else
{
return 0;
}
}
```

15.5.2. Использование модуля DLL

Большинство модулей DLL загружается неявно, и их загрузкой и выгрузкой управляет Windows. Процедура поиска библиотек, загружаемых подобным образом, аналогична процедуре поиска выполняемых файлов: сначала поиск происходит в папке, из которой было загружено использующее их приложение, затем выполняется поиск в текущей папке, затем в папке Windows\System, затем в папке Windows и, наконец, в каждой из папок, указанных в переменой PATH.

Обычно при установке приложения модули DLL принято помещать в папку Windows или Windows\System. Но в процессе разработки приложения в качестве временного хранилища для этой цели можно использовать папку выполняемых файлов проекта. Однако нужно следить за тем, чтобы в итоге в разных папках, включая папки Windows, Windows\System и папку проекта, не оказались разные версии одного и того же модуля DLL.

Неявная загрузка и использование модуля DLL осуществляются так же просто, как и работа с обычными функциями. При компиляции модуля DLL транслятор Microsoft Visual C++ создает файл с расширением .lib. (Таким образом помимо файла DISKFREE.dll существует еще и файл DISKFREE.lib, созданный компилятором.) Файл библиотеки (.lib) используется для разрешения адресов загрузки в модуле DLL и содержит полное имя динамически подключаемой библиотеки, тогда как файл заголовка содержит его описание.

Необходимо подключить файл заголовка к исходному файлу, использующему функции из модуля DLL, и указать имя файла библиотеки (.lib) в поле Additional Dependencies окна Linker страницы свойств проекта (рис. 15.1).

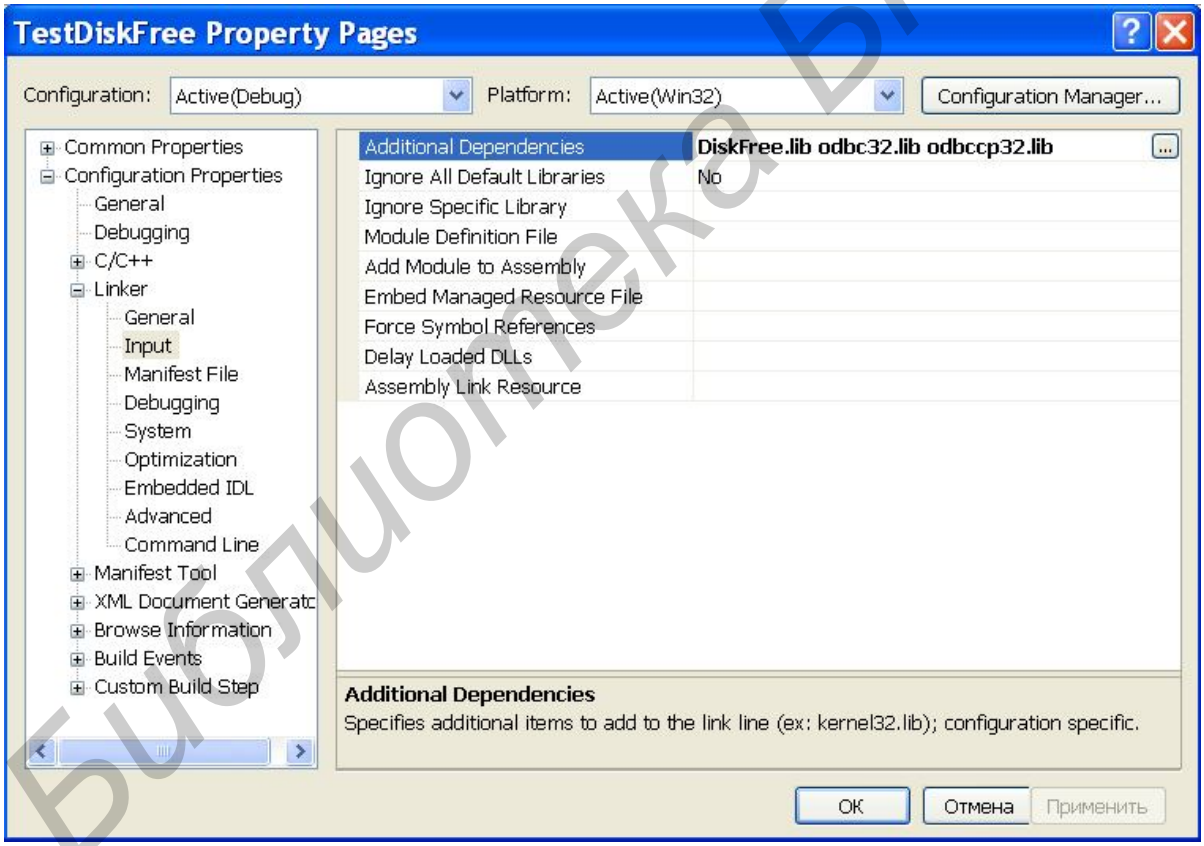

Рис. 15.1. Настройка свойств проекта

Для проверки работы модуля DiskFree создайте консольное приложение с именем TestDiskFree и добавьте в него файл TestDiskFree.cpp исходного текста на C++. Внесите в файл текст программы, представленный в листинге 15.3.

Скопируйте в папку этого проекта файл DiskFree.h и добавьте его к проекту. Скопируйте в папку TestDiskFree файлы DiskFree.dll и DiskFree.lib (они находятся в папке DiskFree\Debug). Выполните все указанные выше изменения в установках проекта и скомпилируйте его.

Листинг 15.3.

```
#include <afx.h>
#include <iostream.h>
#include "diskfree. h"
#define CodeTrace(arg) \
cout << #arg << endl;\
arg
int main()
{
CodeTrace( cout << DiskFree(2) << endl );
return 0;
}
```

Эта программа неявно загружает модуль DLL посредством подключения файла diskfree.h, а затем использует его в работе. При выполнении программы макрос CodeTrace просто печатает оператор программы перед его выполнением. Фактически вся работа приложения состоит в вызове функции DiskFree() с целью определения имеющегося свободного дискового пространства на диске 2. Диск 0 – это A:, диск 1 – B: и диск 2 – C:. Результаты работы скомпилированной и запущенной программы представлены ниже.

```
Cout << DiskFree(2) << endl
36258777
```

Согласно программе TestDiskFree на диске C: компьютера, использованного для запуска этой программы, находится почти 37 Мбайт свободного дискового пространства.

16. Введение в технологии OLE и Active X

16.1. Понятие составных документов

Рассмотрим теорию и концепции технологии ActiveX, которая построена на основе модели COM (Component Object Model – модель многокомпонентных объектов). До недавнего времени технология, построенная на основе COM, носила название OLE (Object Linking and Embedding – связывание и внедрение объектов), но теперь она обозначается термином ActiveX. Наличие Developer Studio и MFC делает использование технологии ActiveX намного проще, поскольку они выполняют большую часть черновой работы, оставляя ее невидимой для разработчика.

Windows всегда обеспечивала пользователям возможность одновременного запуска нескольких приложений. И с самого начала существования этой системы программисты хотели иметь на вооружении методы, с помощью которых запущенные приложения могли бы обмениваться информацией в процессе выполнения. Прекрасным инструментом стал буфер обмена Clipboard, но пользователь все еще должен был выполнять вручную многие операции. Механизм DDE (Dynamic Data Exchange – динамический обмен данными) предоставил приложениям возможность обмена данными, но при этом сохранял некоторые серьезные ограничения. Затем появилась технология OLE1. Позднее ее сменила OLE2, с течением времени переименованная Microsoft просто в OLE, и, наконец, теперь она получила название ActiveX.

Технология ActiveX дает возможность пользователю и приложениям ориентироваться на работу с документами, и это, пожалуй, самое главное. Если пользователь решает подготовить годовой отчет с помощью приложений, поддерживающих ActiveX, он может сосредоточиться именно на отчете как таковом. Вероятно, часть этой работы будет выполнена в Word, а еще некоторая ее часть – в Excel, но для пользователя суть дела заключена не в приложениях. Подобная переориентация сейчас происходит во многих направлениях и вызвано это объектно-ориентированным образом мышления большинства программистов. Ныне кажется более естественным разделить работу между несколькими различными приложениями, способными взаимодействовать между собой, чем написать одно гигантское приложение на все случаи жизни.

В основе документо-ориентированного подхода лежит идея составных документов – файлов, созданных несколькими приложениями. Если ваш отчет нуждается в иллюстрациях, вы создаете их с помощью какой-либо графической программы, а затем, когда они будут готовы, помещаете их в текст. Если в годовой отчет необходимо включить таблицу, данные для которой уже введены в электронную таблицу, не следует заново вводить и упорядочивать их, используя средства табличной обработки текстового редактора,

или даже импортировать их. Поместите их в текст отчета непосредственно как фрагмент электронной таблицы.

Создание составных документов возможно путем внедрения и связывания. При связывании в основном документе сохраняется лишь информация о расположении связанного файла. Связывание в документах целесообразно в том случае, если планируется использовать связываемый файл во многих документах. При этом изменения, вносимые в файл, будут автоматически отражаться во всех документах, с которыми он был связан. При внедрении создается копия объекта, которая и помещается в составной документ. Если в дальнейшем откорректировать исходный файл, изменения не будут отражены в составном документе.

Внедрение объектов в документ следует проводить в том случае, если вы планируете создать составной документ и в дальнейшем работать над ним как над единым целым, не обращаясь более к файлам отдельных составляющих его частей. Любые выполненные в документе изменения не повлияют на остальные файлы на диске, в том числе и на те, копии которых были помещены в составной документ. Внедрение значительно увеличивает размеры файлов документов.

Чтобы выполнить внедрение или связывание двух объектов, необходимо иметь контейнер и сервер. Контейнером является приложение, в документ которого объект внедряется или с документом которого объект связывается (в приведенных выше примерах это Word). Сервером является приложение, в котором объект был создан и которое может быть запущено, когда на объекте будет сделан двойной щелчок (в нашем случае это Excel).

Сервер и контейнер дают пользователю возможность создавать именно такие документы, в которых он нуждается. Данная концепция является значительным шагом в направлении создания блочного программного обеспечения и утверждения документо-ориентированного подхода в работе. Но технология ActiveX включает в себя помимо связывания и внедрения еще и многое другое.

16.2. Возможности AppWizard по созданию приложений ActiveX

Настройка будущего проекта на работу с составными документами осуществляется на этапе создания приложения с помощью ApplicationWizard – выбор уровня поддержки операции с составными документами «Compound Document Support». Окно MFC ApplicationWizard при этом будет выглядеть так, как показано на рис. 16.1.

На выбор предлагается пять вариантов поддержки:

– если не планируется создание OLE-приложения, выберите переключатель None (Никакой);

– если вы хотите, чтобы в приложении использовались связанные или внедренные объекты OLE (например такие, как документы Word или рабочие листы Excel), выберите переключатель Container (Контейнер);

– если планируется создание приложения, документы которого могли бы быть внедрены в другое приложение, но при этом само приложение не будет использоваться автономно, выберите переключатель Mini server (Мини-сервер);

– если ваше будущее приложение будет не только служить сервером для других приложений, но и сможет работать автономно, выберите переключатель Full server (Полный сервер);

– если создаваемое приложение должно обладать способностью включать документы других приложений и само обслуживать их своими объектами, выбирайте переключатель Container/Full server(и контейнер, и сервер).

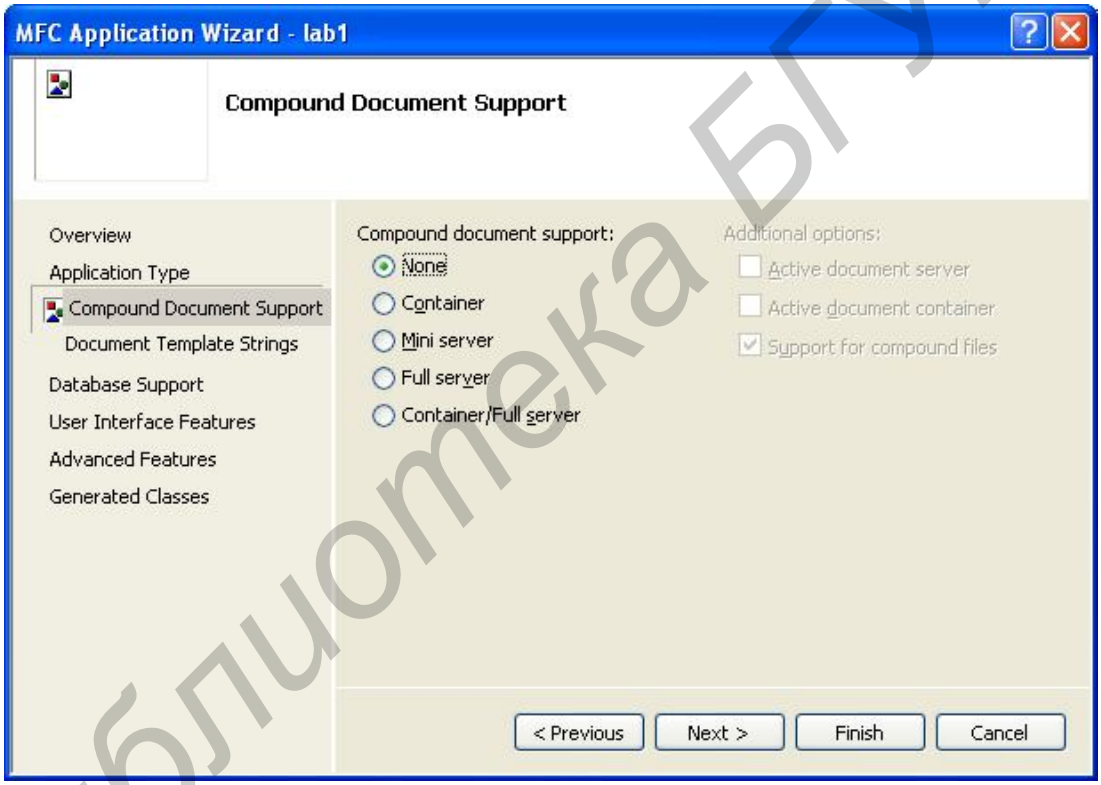

Рис. 16.1. Настройка на работу с составными документами

Если вы выбрали какой-либо из вариантов поддержки составных документов, то придется поддерживать и составные файлы (compound files). Составные файлы содержат один или более объектов ActiveX и сохраняются на диске в особом формате, так что один из объектов может быть заменен без переписи всего файла. Таким образом удается сберечь довольно много времени. В диалоговом окне имеется группа переключателей: Support for compound files (если необходима поддержка составных файлов), Active document server и Active document container.

16.3. Основные понятия COM

Составные документы – лишь частный случай более общей проблемы: как разные программные компоненты должны предоставлять друг другу сервисы. Для решения этой проблемы архитекторы OLE создали группу технологий, область применения которых гораздо шире составных документов. Основу OLE 2 составляет важнейшая из этих технологий – модель многокомпонентных объектов (Component Object Model – COM). Новые возможности многим обязаны COM, предоставившей общую парадигму взаимодействия программ любых типов: библиотек, приложений, системного программного обеспечения и др.

Подход, предложенный COM, можно использовать при реализации практически любой программной технологии, и его применение дает немало существенных преимуществ. Каким образом одна часть программного обеспечения должна получать доступ к сервисам, предоставляемым другой частью? Ответ зависит от того, что представляют собой эти части. Приложения, скомпонованные с библиотекой, могут пользоваться ее сервисами, вызывая функции из этой библиотеки. Приложение также может использовать сервисы другого приложения, являющегося совершенно отдельным процессом. В этом случае два таких локальных процесса взаимодействуют посредством некоего механизма связи, который обычно требует определения протокола между этими приложениями (набор сообщений, позволяющий одному приложению выдавать запросы, а другому соответствующим образом отвечать на них). Еще пример – приложение, использующее сервисы операционной системы. Здесь приложение обычно выполняет системные вызовы, обрабатываемые операционной системой. Наконец, приложению могут понадобиться сервисы, предоставляемые программным обеспечением, выполняемым на другой машине, доступ к которой осуществляется по сети. Получить доступ к таким сервисам можно множеством способов, таких как обмен сообщениями с удаленным приложением или вызовы удаленных процедур.

В принципе проблема одна: одна часть программного обеспечения должна получить доступ к сервисам, предоставляемым другой частью. Но в каждом отдельном случае механизм доступа разный: вызовы локальных функций, передача сообщения средствами связи между процессами, системные вызовы (которые с точки зрения программиста выглядят практически так же, как и вызовы функций) или какая-то разновидность сетевых коммуникаций.

COM определяет стандартный механизм, с помощью которого одна часть программного обеспечения предоставляет свои сервисы другой и который работает во всех описанных выше случаях. Каждый такой объект поддерживает один или несколько интерфейсов, состоящих из методов. Метод – это функция или процедура, которая выполняет некоторое действие и может быть вызвана программным обеспечением, использующим данный объект (клиентом объекта). Методы, составляющие каждый из интерфейсов, обычно

определенным образом взаимосвязаны. Клиенты могут получить доступ к сервисам объекта COM только через вызовы методов интерфейсов объекта – у них нет непосредственного доступа к данным объекта.

Представим себе, например, корректор орфографии, реализованный в виде объекта COM. Такой объект может поддерживать интерфейс, включающий методы типа «НайтиСлово», «ДобавитьКСловарю» и «УдалитьИзСловаря». Если позднее разработчик объекта COM захочет добавить к этому объекту поддержку словаря синонимов, то объекту потребуется еще один интерфейс, возможно, с единственным методом вроде «НайтиСиноним». Методы каждого из интерфейсов сообща предоставляют связанные друг с другом сервисы: либо корректировку правописания, либо доступ к словарю синонимов.

Сам объект всегда реализуется внутри некоторого сервера. Сервер может быть либо динамически подключаемой библиотекой (DLL), подгружаемой во время работы приложения, либо отдельным самостоятельным процессом.

Чтобы вызывать методы интерфейса объекта COM, клиент должен получить указатель на этот интерфейс. Обычно COM-объект предоставляет свои сервисы посредством нескольких интерфейсов, и клиенту требуется отдельный указатель для каждого интерфейса, методы которого он намерен вызывать. Например, клиенту нашего простого объекта COM понадобился бы один указатель интерфейса для вызова методов интерфейса корректора орфографии и другой – для вызова методов интерфейса словаря синонимов.

Любой COM-объект – это экземпляр определенного класса. Объекты одного класса могут, например, реализовывать сервисы корректировки орфографии и словаря синонимов, тогда как объекты другого класса – представлять банковские счета.

Обычно знать класс объекта необходимо для запуска экземпляра этого объекта, выполняемого с помощью библиотеки COM. Эта библиотека COM присутствует на любой системе, поддерживающей COM, и имеет доступ к справочнику всех доступных на данной машине классов COM-объектов. Клиент может, например, вызвать функцию библиотеки COM, передав ей класс нужного ему COM-объекта и задав один из поддерживаемых объектом интерфейсов, указатель которого нужен клиенту в первую очередь. Эти сервисы реализованы библиотекой COM в виде обычных вызовов функций, а не через методы интерфейса COM.

Затем библиотека COM запускает сервер, реализующий объекты данного класса. Кроме того, библиотека возвращает клиенту указатель требуемого интерфейса вновь созданного экземпляра объекта. Далее клиент может запросить указатели на другие необходимые ему интерфейсы непосредственно у самого объекта. Получив указатель на нужный ему интерфейс выполняющегося объекта, клиент может использовать сервисы объекта, просто вызывая методы этого интерфейса.

В большинстве объектных технологий объект поддерживает только один интерфейс с одним набором методов. А вот COM-объекты могут – и почти всегда это делают – поддерживать более одного интерфейса. Например, у C++-объекта лишь один интерфейс, включающий в себя все методы объекта. COM-объект с его несколькими интерфейсами может быть отлично реализован с несколькими объектами C++ – по одному на каждый интерфейс COM-объекта.

Компонентный подход COM находит различные применения в технологиях Microsoft для Интернета и WWW. Например, средство просмотра WWW фирмы Microsoft Internet Explorer активно использует расширение технологии составных документов OLE – документы ActiveX (ActiveX Documents). Благодаря этому расширению пользователь может просматривать информацию разного типа в дополнение к обычным страницам HTML (Hypertext Markup Language).

17. Обзор технологий ActiveX и OLE

Широкий набор технологий OLE и ActiveX разработан с использованием COM. Ниже кратко представлены наиболее важные технологии COM.

17.1. Автоматизация

Электронные таблицы, текстовые процессоры и другие программы предоставляют широкий спектр полезных возможностей. Чтобы воспользоваться этими возможностями, приложения должны предоставлять свои сервисы не только человеку, но и программам – они должны быть программируемыми. Обеспечение программируемости и является целью автоматизации (Automation, первоначально называвшейся OLE-автоматизацией).

Приложение можно сделать программируемым, обеспечив доступ к его сервисам через обычный COM-интерфейс. Однако так поступают редко. Вместо этого доступ к сервисам приложений осуществляется через диспинтерфейсы (IDispatch). Они очень похожи на интерфейсы (у них есть методы, клиенты осуществляют к ним доступ через указатель интерфейса).

17.2. Перманентность

Объекты состоят из методов и данных, и многим объектам необходимо сохранять свои данные в течение периодов неактивности. Объекту нужно сделать свои данные перманентными (persistent), что обычно означает запись их на диск. COM-объекты достигают этого разными путями. Один из наиболее широко применяемых – структурированное хранилище (Structured Storage).

Традиционные файловые системы обеспечивают совместное использование приложениями одного дискового устройства без конфликтов между ними. Каждое приложение работает со своими собственными файлами и, может быть, даже с собственными подкаталогами независимо от того, чем заняты в тот же момент другие приложения. Приложениям не требуется взаимодействовать друг с другом, чтобы сохранить свои данные, т. к. у каждого есть отдельная область для хранения.

Так как COM обеспечивает совместную работу разных типов программ с помощью одной модели, то независимо разработанный COM-объект может стать частью чего-то, что пользователь считает одним приложением, и в то же время объекту по-прежнему необходимо хранить свои данные на диске отдельно, поэтому используется способ совместного использования одного файла несколькими COM-объектами. Такую возможность и предоставляет структурированное хранилище. Создавая, по сути дела, файловую систему внутри каждого файла, структурированное хранилище предоставляет каждому компоненту, составляющему некоторое приложение, собственный отдельный кусок пространства хранилища, собственные «файлы». С точки зрения пользователя файл только один. Однако с точки зрения приложения каж-

дый компонент имеет собственную область для хранения данных и все такие области находятся внутри одного дискового файла.

Чтобы это реализовать, структурированное хранилище определяет два типа COM-объектов, каждый из которых поддерживает соответствующие интерфейсы. Эти объекты известны как хранилища (storage) и потоки (streams) и аналогичны, соответственно, каталогам и файлам обычной файловой системы. Файл структурированного хранилища может содержать данные многих COM-объектов, каждый из которых использует для сохранения своих данных собственное хранилище или поток. Точно так же, как обычная файловая система обеспечивает совместное использование диска несколькими приложениями, структурированное хранилище позволяет разным приложениям сообща использовать один файл.

Моникер (moniker, имя, кличка) сам по себе является COM-объектом, но весьма специфического назначения: любой моникер знает, как создать и инициализировать экземпляр другого объекта. Например, имея моникер для банковского счета, можно попросить его создать счет, инициализировать его и соединить с ним. Все детали, необходимые для выполнения этих действий, скрыты от клиента. Если он хочет работать посредством моникеров с двумя банковскими счетами, то ему потребуется два отдельных моникера по одному для каждого объекта – счета.

17.3. Единообразная передача данных и объекты с подключением

Стандартный способ обмена информацией в мире COM – единообразная передача данных (Uniform Data Transfer). Приложения, использующие этот способ, должны поддерживать определенные интерфейсы COM. Методы этих интерфейсов определяют стандартные способы для описания передаваемых данных, для указания их местоположения и собственно для их пересылки. Они даже определяют простой механизм, позволяющий одному приложению уведомить другое о том, что нужные последнему данные стали доступны.

Полезная в определенных ситуациях простая схема, определенная как единообразная передача данных для уведомления клиента о наличии интересующих его данных, не вполне достаточна. Именно для ликвидации этих недостатков на основе COM была разработана технология объектов с подключением (Connectable Objects). Обеспечивая более общий механизм обратной связи объекта с клиентом, объекты с подключением позволяют клиенту легко получать уведомления об интересующих его событиях.

17.4. Составные документы

В текстовые процессоры добавляются графические возможности, в электронные таблицы – средства построения диаграмм, и, представляется, все кончится созданием одного большого приложения для решения всех задач. Но в действительности цель как раз не в этом, а в интеграции разных

приложений. Например, добавлять поддержку графики в текстовый процессор не потребуется, если внутри него можно будет использовать некоторое уже существующее графическое приложение. Пользователю должно представляться нечто такое, что выглядит как один документ, хотя на самом деле над разными частями такого документа совместно работают разные приложения.

Для решения этой проблемы предназначена технология OLE (ранее известная как документы OLE – OLE Documents). Поддерживая нужные COM-объекты, каждый с собственным набором интерфейсов, независимые приложения могут совместно работать, чтобы пользователь получил один составной документ. Все эти интерфейсы носят абсолютно общий характер — ни одно приложение не знает, что представляют собой другие. OLE поможет просто задействовать в случае необходимости существующее приложение электронной таблицы.

Определенный OLE стандартный интерфейс обеспечивает взаимодействие между приложениями любых типов и любых производителей, а не только между электронными таблицами и текстовыми процессорами Microsoft.

При создании составного документа с помощью OLE одно из приложений всегда является контейнером. Как следует из названия, контейнер определяет самый общий документ, в котором содержится все остальное. Другие приложения – серверы – могут размещать свои документы внутри документа-контейнера.

При использовании OLE документ сервера может быть либо связан, либо внедрен в документ контейнера. Связанный документ сервера хранится в отдельном файле, а в документе контейнера хранится лишь связь с этим файлом. На самом деле связью является моникер. Внедренный документ сервера хранится в том же файле, что и документ контейнера. Два приложения при этом совместно используют общий файл с помощью структурированного хранилища.

17.5. Управляющие элементы ActiveX

Почему для включения в текстовый документ электронной таблицы необходимо использовать весь Excel? Если нужны только основные возможности электронных таблиц, то, вероятно, можно обойтись меньшим, более быстрым и дешевым компонентом. Многим программистам понравилась бы возможность построить целое приложение по большей части из готовых компонентов.

Именно подобное желание и привело к идее компонентного программного обеспечения – области, где COM способен на очень многое. Повторно применимые компоненты можно создавать на основе исключительно самой COM, но для этой цели полезно определить и некоторые стандартные интерфейсы и соглашения. Используя последние, можно создавать компоненты,

единообразно выполняющие такие распространенные задачи, как обеспечение пользовательского интерфейса и посылка сообщений клиенту. Эти стандарты и определяет спецификация управляющих элементов ActiveX (ActiveX Controls).

Управляющий элемент ActiveX – независимый программный компонент, выполняющий специфические задачи стандартным способом. Разработчики могут задействовать один или несколько таких элементов в приложении, чтобы получить преимущества функциональных возможностей существующего программного обеспечения.

Первоначально управляющие элементы ActiveX были известны под названием «управляющие элементы OLE или OCX». Microsoft изменила название, чтобы отразить некоторые новые возможности, сделавшие эти элементы более подходящими для Internet и WWW. Например, управляющий элемент ActiveX может хранить свои данные на странице где-то в WWW либо может быть выкачан с сервера WWW и затем запущен на машине клиента. И контейнер, в котором работает управляющий элемент, не обязан быть средой программирования – вместо этого он может быть средством просмотра WWW.

Управляющие элементы ActiveX не отдельные приложения. Напротив, они являются серверами, которые подключаются к контейнеру элементов. Как обычно, взаимодействие между управляющим элементом и его контейнером определяется различными интерфейсами, поддерживаемыми COM-объектами.

Фактически управляющие элементы ActiveX используют многие другие технологии OLE и ActiveX. Например, управляющие элементы обычно поддерживают интерфейсы для внедрения и зачастую предоставляют доступ к своим методам через диспинтерфейсы автоматизации.

18. Использование элементов Active X для разработки интерфейса приложения

Одна из главных целей COM – обеспечить производство компонентных программ, т. е. приложений, которые собираются из готовых элементов. Большинство действительно полезных компонентов достаточно сложны. Для облегчения разработки компонентов необходимо определить стандартный способ отображения компонентами собственного пользовательского интерфейса.

Компонентам может потребоваться механизм для посылки событий своим клиентам, а также механизм, позволяющий клиенту читать и изменять свойства компонента. Подобные стандарты значительно облегчат жизнь и программному обеспечению, использующему компоненты.

Определение стандартов для программных компонентов и является задачей спецификации управляющих элементов ActiveX. Спецификация управляющих элементов ActiveX определяет правила создания контейнеров управляющих элементов (control containers) – клиентских программ, знающих как работать с этими элементами. Эти правила рассматриваются с трех точек зрения: конечного пользователя, разработчика приложения, создателя управляющего элемента.

18.1. Точка зрения конечного пользователя

Обычно конечные пользователи не подозревают о том, что они работают с управляющим элементом. Вместо этого они видят графический пользовательский интерфейс: кнопки, которые можно нажимать, ползунки, которые можно двигать, поля, куда можно вводить текст, и т.д.

Большинство современных операционных систем позволяют приложениям представить подобный интерфейс. То, что пользователь видит единым целым, на самом деле является контейнером с управляющими элементами ActiveX.

Контейнер управляющих элементов подобен контейнеру составных документов OLE, но поддерживает несколько дополнительных интерфейсов для работы с управляющими элементами ActiveX. Каждый управляющий элемент подключен к контейнеру и обычно представляет свой собственный пользовательский интерфейс как внедренный объект, поддерживающий активизацию «на месте».

Например, кнопка на экране может быть пользовательским интерфейсом некоего управляющего элемента ActiveX. Щелкая ее и взаимодействуя с исполняющимся в результате кодом, пользователь фактически активизирует элемент управления ActiveX.

То что пользователь видит как одно приложение, представляющее один интегрированный пользовательский интерфейс, на самом деле – контейнер

управляющих элементов, полный различных дискретных управляющих элементов ActiveX, каждый из которых выполняет часть общей работы.

18.2. Точка зрения разработчика приложения

Построение приложения из управляющих элементов заметно отличается от создания приложения с нуля. Чтобы при создании приложения использовать управляющие элементы, разработчик вначале должен принять решение относительно их контейнера.

В числе популярных инструментов создания контейнеров Microsoft Visual Basic и Developer Studio и множество других средств третьих фирм. Кроме того, в качестве контейнера можно использовать средство просмотра WWW, при этом управляющий элемент должен быть встроен в HTML-файл.

Затем следует решить, какие управляющие элементы ActiveX включить для обеспечения предполагаемых функциональных возможностей. В состав Visual Basic и Developer Studio входит множество управляющих элементов, так что программист без труда найдет нужный. Если нет, то проблему решит большой и быстро растущий рынок управляющих элементов ActiveX третьих фирм, где представлена продукция сотен компаний.

Если подходящего управляющего элемента нет ни в Visual Basic, ни на рынке третьих фирм, его можно разработать самостоятельно. Для этого необходима совершенно иная квалификация, чем для создания приложения, использующего управляющий элемент.

Для создания приложения разработчик, выбрав управляющий элемент, задает его местоположение и размеры на бланке (или шаблоне диалогового окна). Практически каждый управляющий элемент ActiveX определяет некоторый набор свойств (property). Задавая значения этих свойств, разработчик может изменить цвет управляющего элемента, цвет и толщину рамки вокруг него и т. д.

В дополнение к свойствам управляющий элемент обычно определяет набор событий (events). Посылка и прием этих событий осуществляется с помощью механизма на основе COM, но как именно это происходит, для разработчика, собирающего данное приложение, значения не имеет.

Обычно контейнеры управляющих элементов позволяют программисту задать действие (в виде кода функции или метода), которое должно быть выполнено в ответ на сообщение, полученное от управляющего элемента.

Кроме набора свойств и набора событий у типичного управляющего элемента есть и методы. У клиентов элементов ActiveX – контейнеров управляющих элементов – имеется возможность выдачи запросов. Что именно может запросить контейнер у управляющего элемента, определяется методами, поддерживаемыми последним.

Доступ к методам управляющих элементов ActiveX всегда осуществляется через диспинтерфейсы IDispatch. С точки зрения разработчика приложе-

ния вызов методов управляющего элемента ActiveX осуществляется точно так же, как и вызов методов диспинтерфейса любого COM-объекта.

Таким образом, для разработчика приложения три элемента: свойства, события и методы определяют возможности управляющего элемента ActiveX.

18.3. Точка зрения создателя управляющего элемента

То, как выглядит управляющий элемент для своего создателя, зависит от используемого инструментария. Написать новый управляющий элемент «с нуля» не так-то просто.

Простой управляющий элемент, поддерживающий ограниченный набор возможностей, лучше всего, вероятно, разработать целиком вручную – так можно получить более быстрый и компактный код.

Но для реализации сложного управляющего элемента необходим мощный набор инструментов. В состав Microsoft Developer Studio входит Control Development Kit (CDK – набор инструментов для разработки управляющих элементов) и ControlWizard (мастер управляющих элементов), сочетание которых позволяет опытным программистам на C++ разрабатывать управляющие элементы, даже не зная большей части деталей реализации управляющих элементов.

Функциональность, определяемая спецификацией управляющих элементов ActiveX, распадается на 4 основных аспекта, для реализации каждого предназначена особая группа интерфейсов:

– обеспечение пользовательского интерфейса;
– обеспечение вызова методов управляющего элемента контейнером;
– посылка событий контейнеру;
– получение информации о свойствах среды контейнера;
– обеспечение доступа к свойствам управляющего элемента и их модификации.

19. Использование технологии OLE DB для доступа к базе данных

OLE DB – набор COM-интерфейсов, предоставляющих приложению единообразный доступ к данным самых различных источников независимо от их местонахождения или типа. Открытая спецификация OLE DB основана на технологии ODBC, она предоставляет открытый стандарт доступа к данным любого типа. ODBC создавалась для взаимодействия с реляционными БД, а OLE DB разрабатывалась как для реляционных, так и для нереляционных источников, включая (но не ограничиваясь) БД на мейнфреймах, серверах и персональных компьютерах, а также хранилища файлов и сообщения электронной почты, электронные таблицы, инструментальные средства управления проектами и пользовательские объекты.

19.1. Применение технологии ADO для доступа к базе данных

ADO реализует высокопроизводительный и удобный прикладной интерфейс для OLE DB. Технология ADO нетребовательна к системным ресурсам, создает минимальную нагрузку на сеть и отличается минимальным числом уровней между приложением и источником данных. ADO предоставляет COM-интерфейс Automation, поэтому она поддерживается любой ведущей инструментальной средой быстрой разработки приложений (Rapid Application Development, RAD), а также другими инструментальными средствами разработки баз данных, приложений и сценариев.

Познакомимся с некоторыми ActiveX-элементами, осуществляющими доступ к базе данных.

ADO Data Control – это графический элемент управления на базе ActiveX с кнопками навигации по записям. Он предоставляет приложению удобный интерфейс для работы с базами данных и позволяет избежать дополнительного кодирования. В ADO Data Control механизм ADO применяется для оперативного создания соединений между поставщиками данных и связанными с данными элементами управления. Элементы управления, связанные с данными, представляют собой ActiveX-элементы пользовательского интерфейса с двумя важными свойствами:

– наличием параметра DataSource, в котором можно задать идентификатор элемента ADO Data Control;

– способностью отображать данные, выбранные связанным с ним элементом ADO Data Control.

Когда элементы управления связаны с ADO Data Control, при просмотре записей все поля отображаются и обновляются автоматически. Такое поведение реализовано в самих элементах и для этого не требуется ни одной дополнительной строчки кода. В Visual C++ есть несколько ActiveX-элементов для работы с данными, например Microsoft DataGrid и Microsoft

DataList. Кроме того, допустимо создавать собственные элементы управления, а также приобретать их у других поставщиков программного обеспечения.

Рассмотрим создание простого приложения, в котором ADO Data Control и DataGrid применяются для отображения записей из таблицы базы данных. Кроме того, мы научимся управлять свойствами этих элементов из исходного кода программы. Перед созданием приложения зарегистрируем источник данных (в нашем случае Users) в операционной системе с помощью Администратора источника данных ODBC.

19.2. Создание приложения UsersADO

Создаем проект MFC AppWizard (.exe) для приложения UsersADO. Тип приложения – SDI. Уровень поддержки баз данных – Data base view without file support. Технология связи – OLE DB, выбор поставщика данных – Microsoft OLE DB Provider for ODBC Drivers (рис. 19.1).

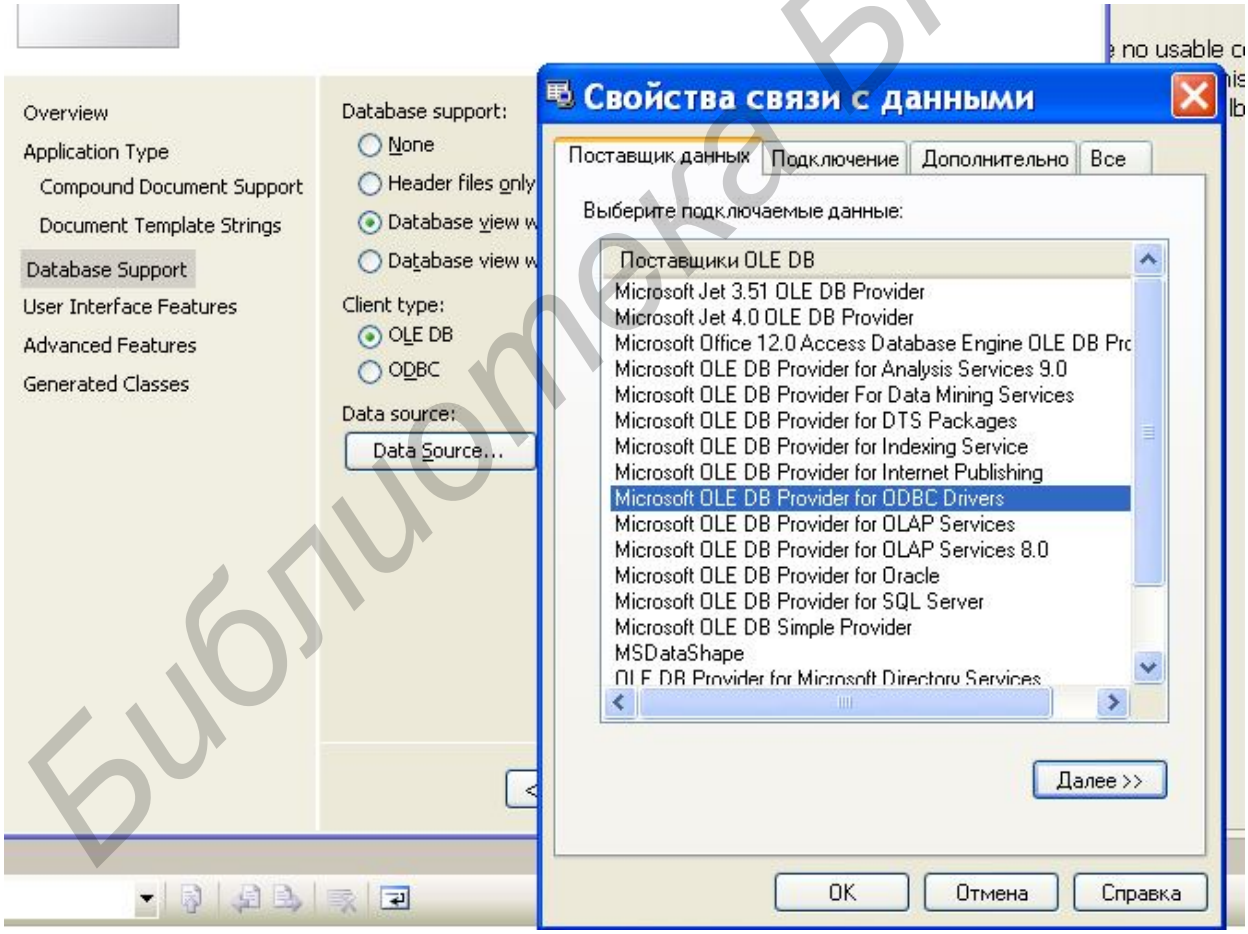

Рис. 19.1. Свойства связи с данными

В закладке «Подключение» выбираем источник данных Users (рис. 19.2), далее настраиваемся на таблицу базы, которая тоже называется Users (рис. 19.3).

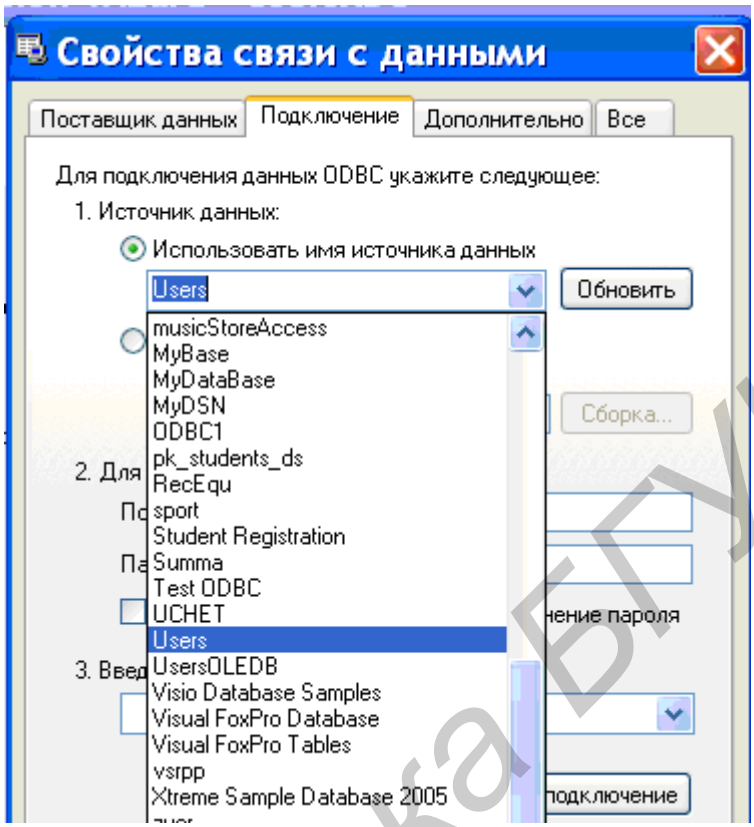

Рис. 19.2. Настройка на источник данных

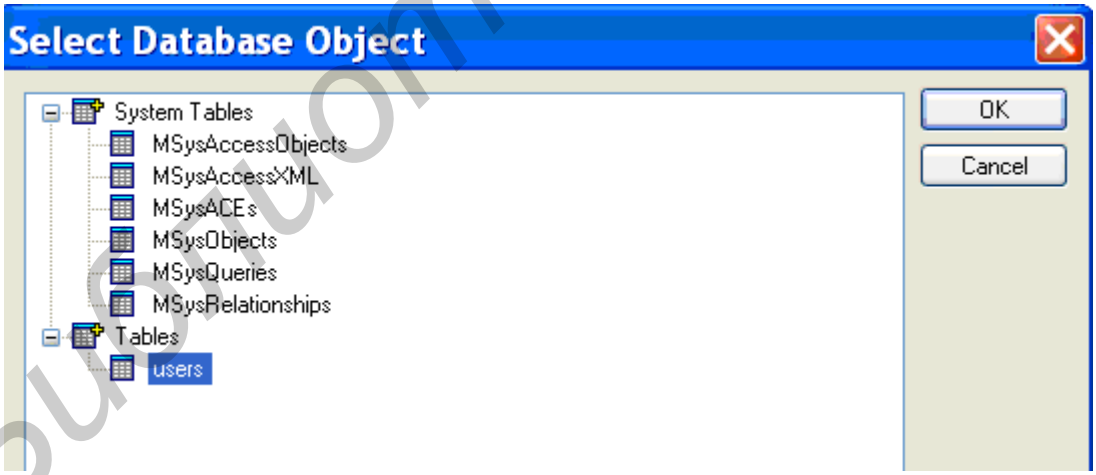

Рис. 19.3. Выбор таблицы базы данных

19.3. Добавление ActiveX-элемента в проект

Добавим в проект ActiveX-элементы для получения выборки из источника данных и ее отображения на форме. Самым простым способом будет

добавление элементов через контекстное меню. Щелкните правой кнопкой мыши на форме, из меню выберите Insert ActiveX Control (рис. 19.4).

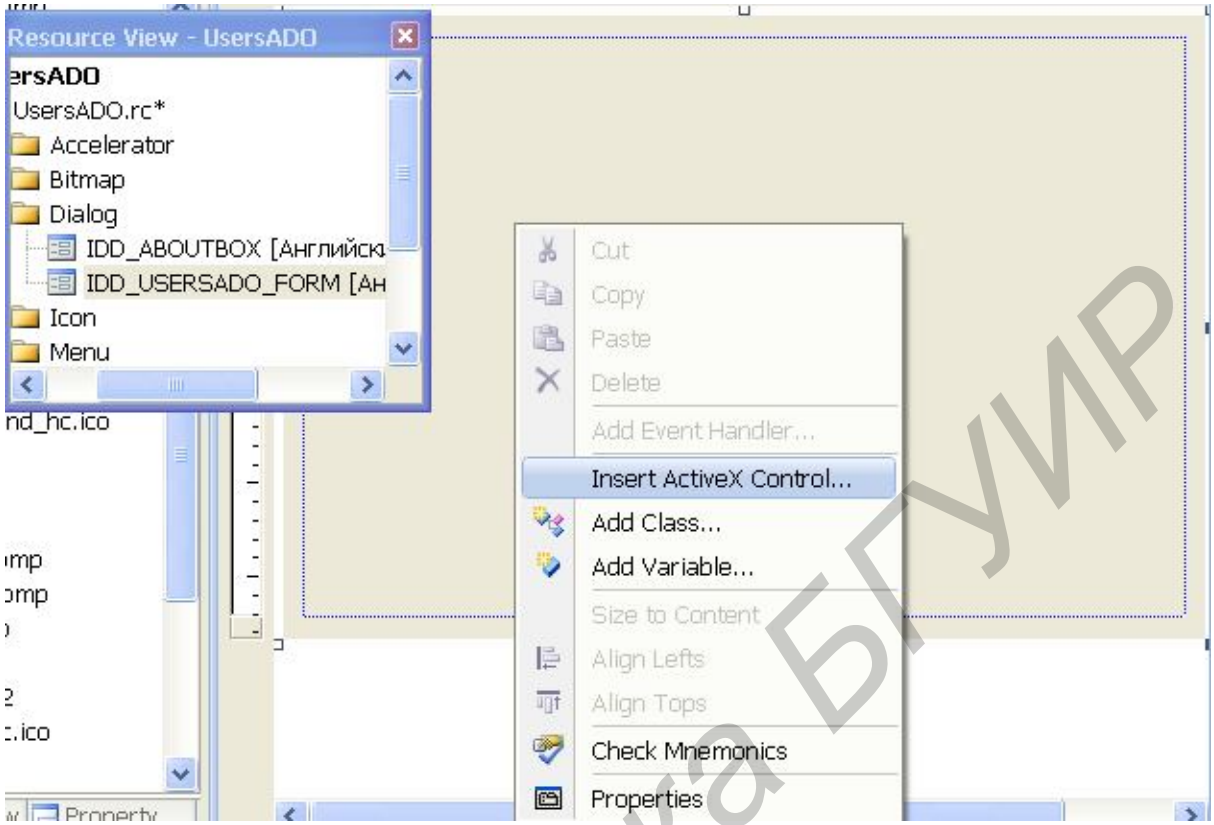

Рис. 19.4. Добавление в проект ActiveX-элементов

Затем в предложенном списке выберем сначала Microsoft ADO Data Control, version 6.0 (OLEDB) (рис. 19.5), затем Microsoft DataGrid Control, Version 6.0 (OLEDB).

Рис. 19.5. Выбор ActiveX-элементов

Наша форма примет вид, представленный на рис. 19.6.

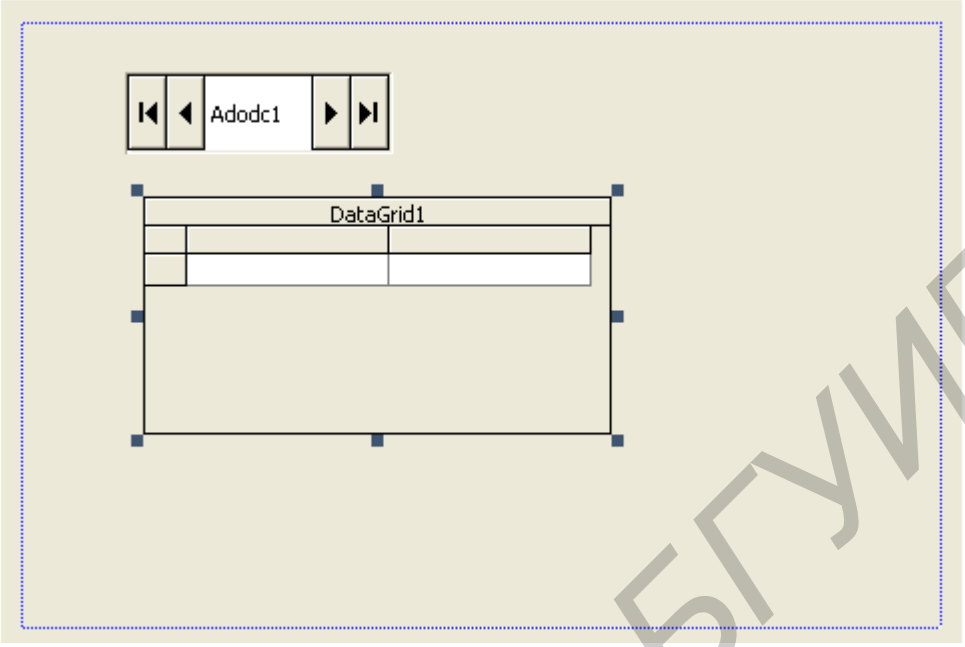

Рис. 19.6. Форма с установленными ActiveX-элементами

В качестве альтернативы первому методу можно сделать следующее: щелкнув правой кнопкой мыши на панели инструментов (ToolBox), выберем из контекстного меню пункт Choose Items… (при желании можно добавить новую закладку в панель инструментов путем выбора пункта Add Tab) (рис. 19.7).

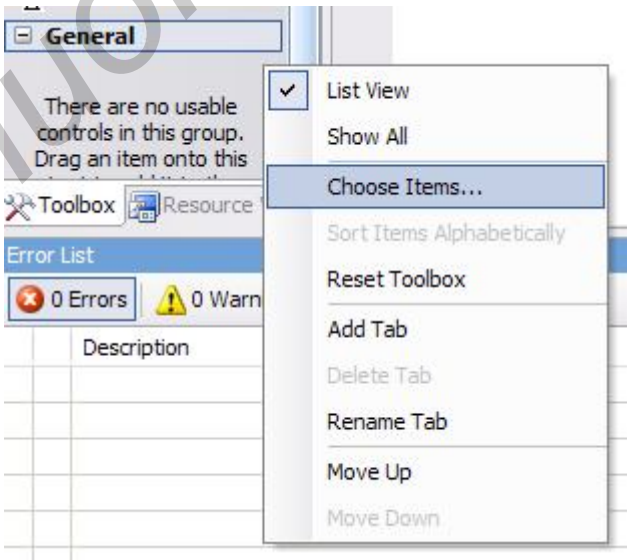

Рис. 19.7. Выбор пункта меню Choose Items…

Затем в появившемся диалоге выберем закладку COM Components и поставим галочки напротив нужных нам элементов (рис. 19.8).

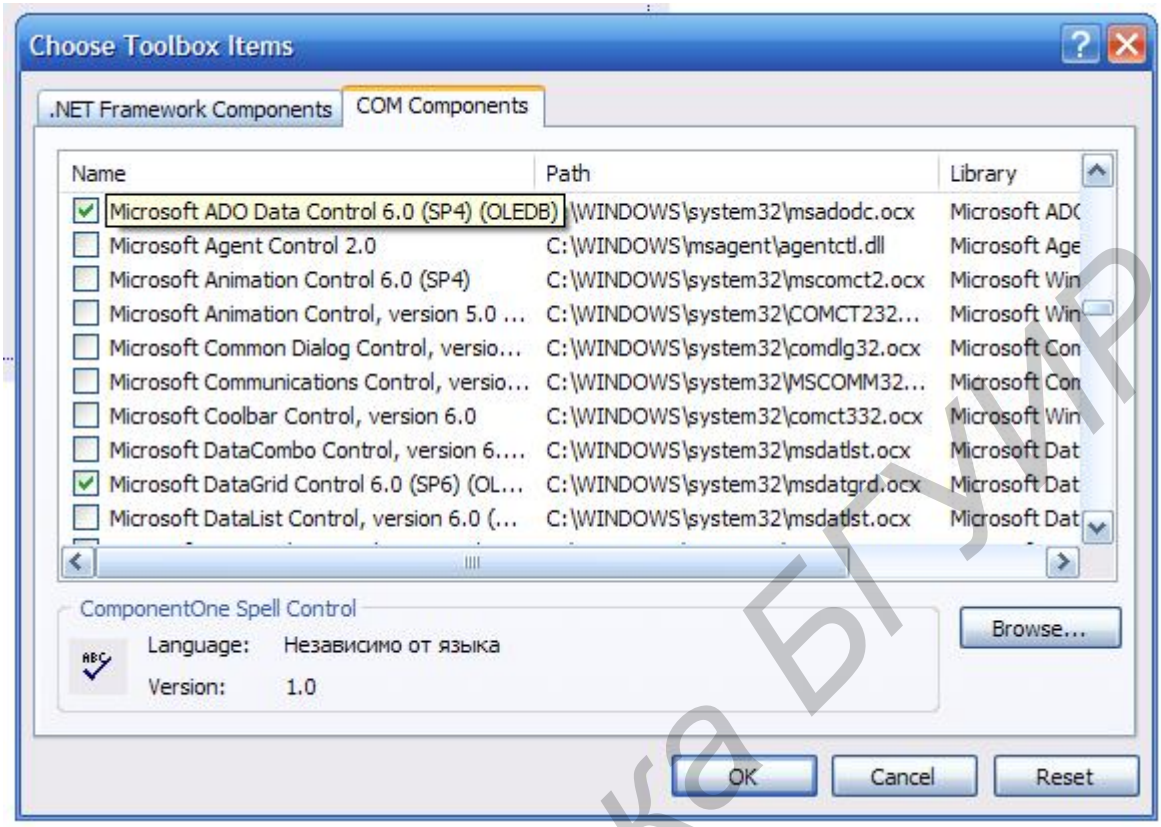

Рис. 19.8. Выбор пункта меню Choose Items…

После нажатия на кнопку OK они появятся на панели инструментов, и появится возможность добавления их на форму путем простого перетаскивания с панели инструментов в нужное нам место (рис. 19.9).

Для редактирования классов нужных нам компонентов, изменения их функциональности, необходимо выполнить следующую последовательность действий.

Выберем в меню пункт Project → Add Class (рис. 19.10). При этом необходимо выйти из режима дизайнера формы, иначе откроется диалог Class Wizard'a.

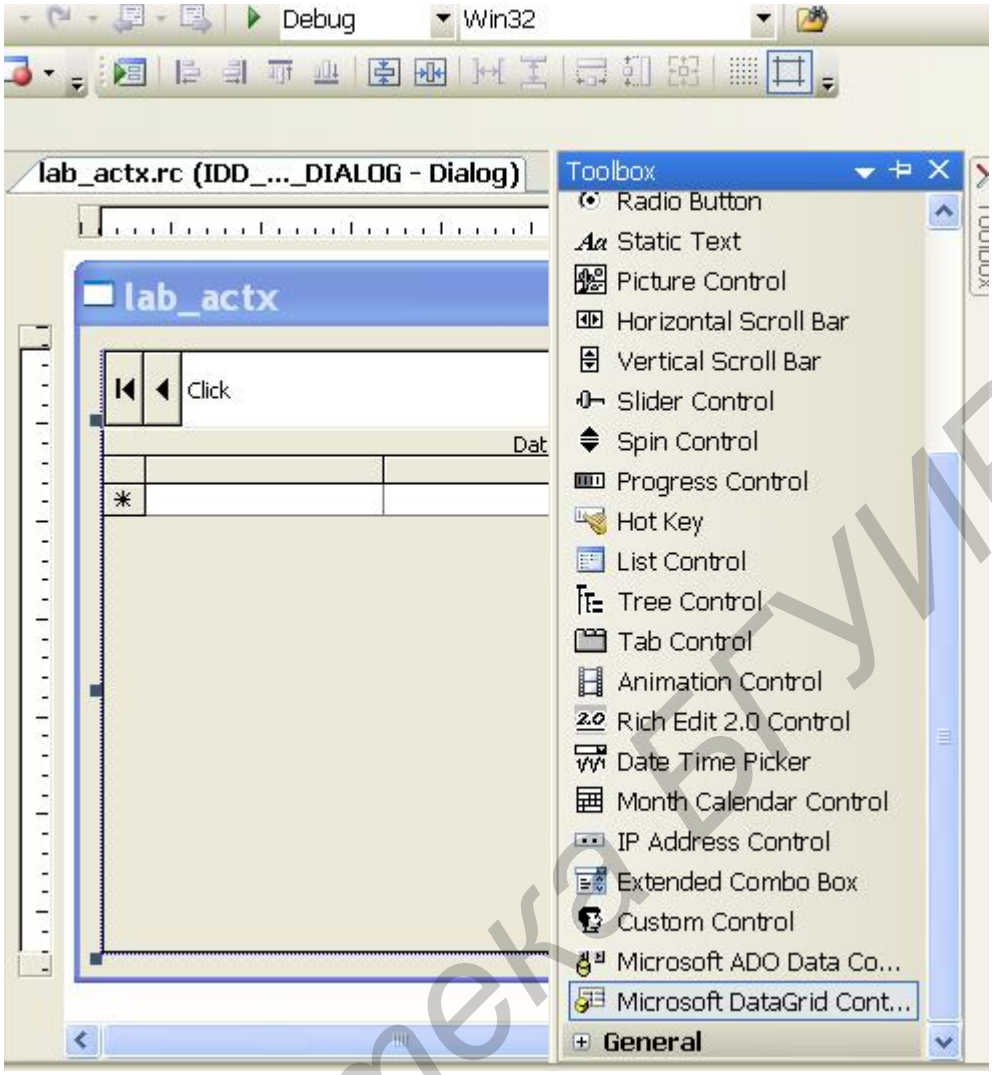

Рис. 19.9. Элементы ActiveX на панели Toolbox

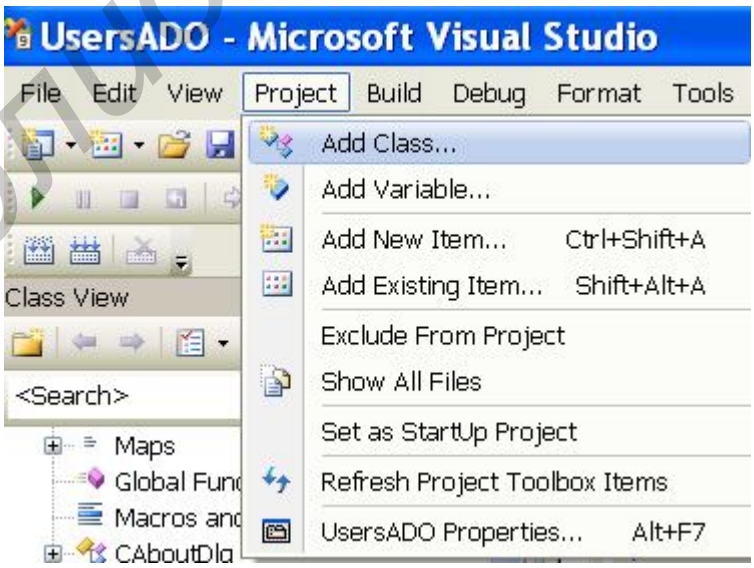

Рис. 19.10. Добавление классов для элементов ActiveX

Выберем этот пункт. В результате появится диалог «Add Class – UsersADO» – диалог выбора категорий добавляемых классов. В этом диалоге выберем по дереву категорий Visual C++ → MFC → MFC Class From ActiveX Control и нажмем кнопку Add (рис. 19.11).

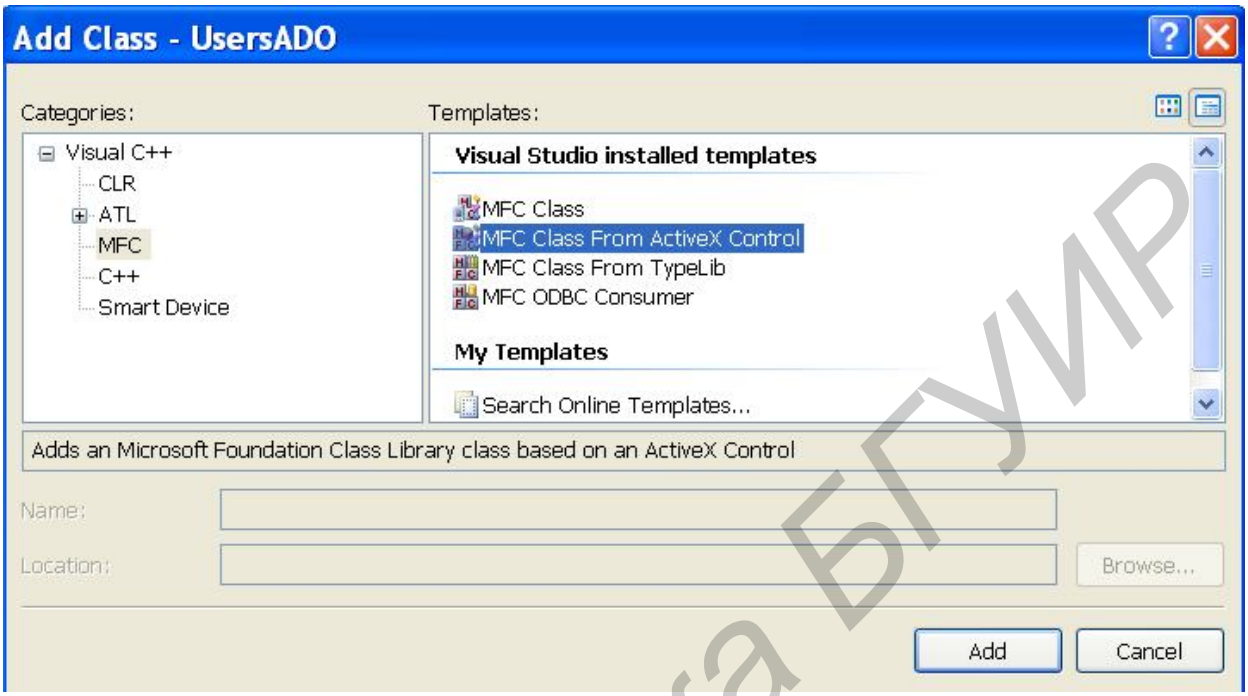

Рис. 19.11. Выбор шаблонов классов для элементов ActiveX

В результате получим диалог добавления классов для элементов ActiveX (рис. 19.12).

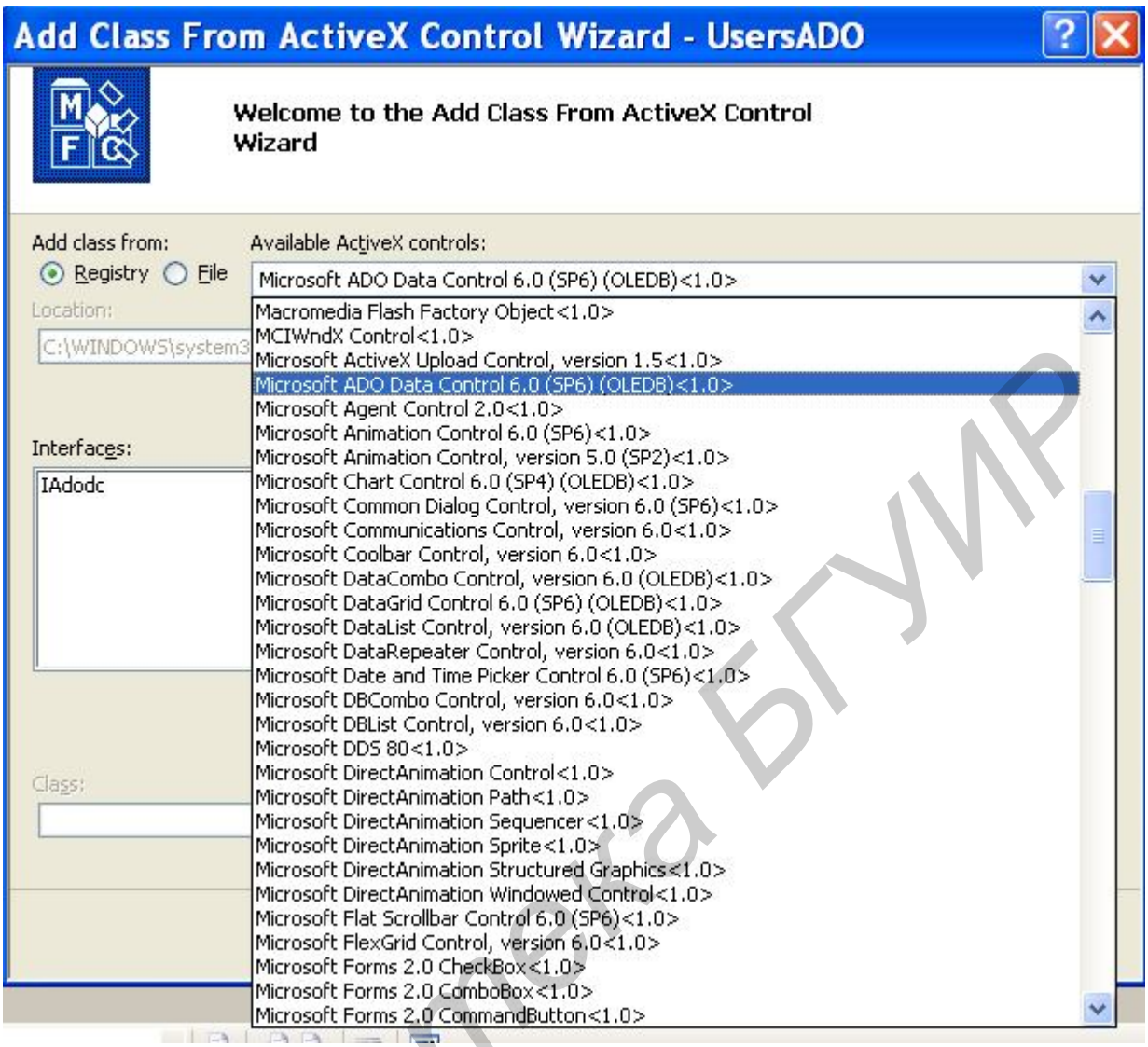

Рис. 19.12. Выбор шаблонов классов для элементов ActiveX

Выберем в списке Available ActiveX Controls нужный нам элемент (см. рис. 19.12). Затем в списке интерфейсов (Interfaces) выберем соответствующий ему интерфейс (для MS ADO Data Control – IAdodc, для MS Data Grid Control – IDataGrid, они будут в списке интерфейсов единственными). Добавим его в список генерируемых классов (Generated Classes) путем нажатия кнопки «>» (либо «>>», в данном случае это не имеет значения, т. к. добавляемые интерфейсы являются единственными в своем списке) (рис. 19.13 и 19.14).

Рис. 19.13. Генерация класса CAdodc

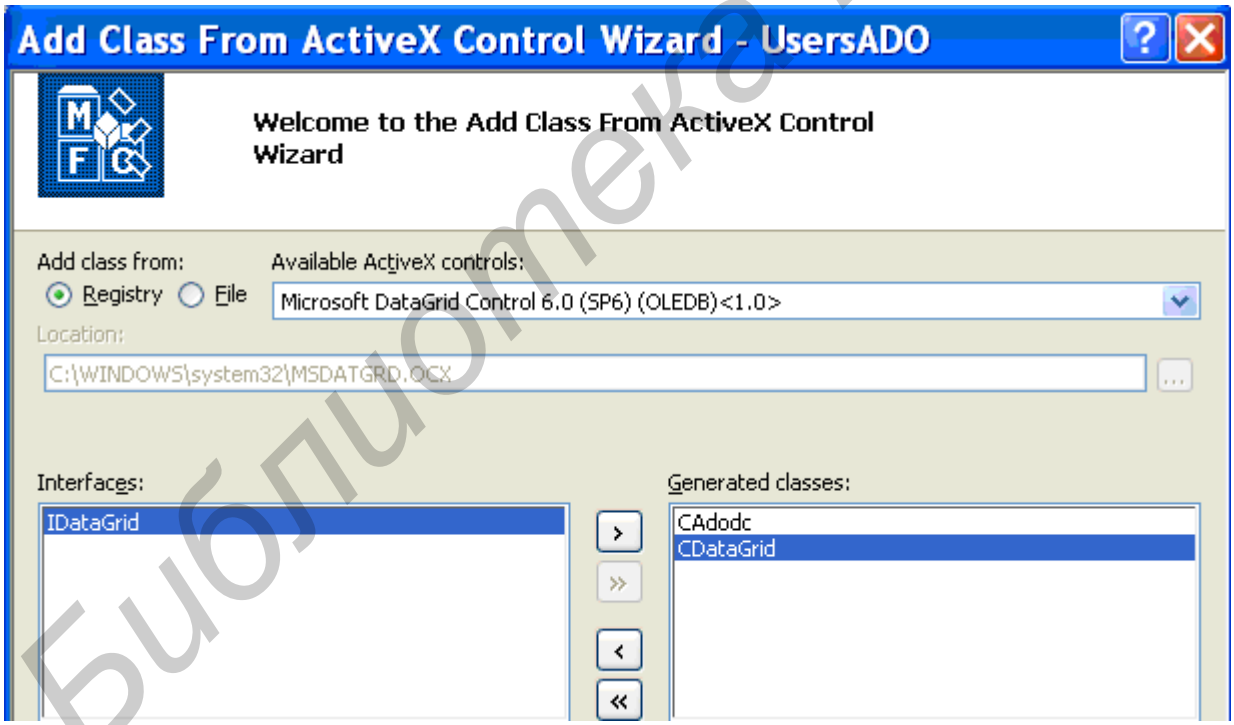

Рис. 19.14. Генерация класса CDaraGrid

Появились новые классы, представленные на рис. 19.15.

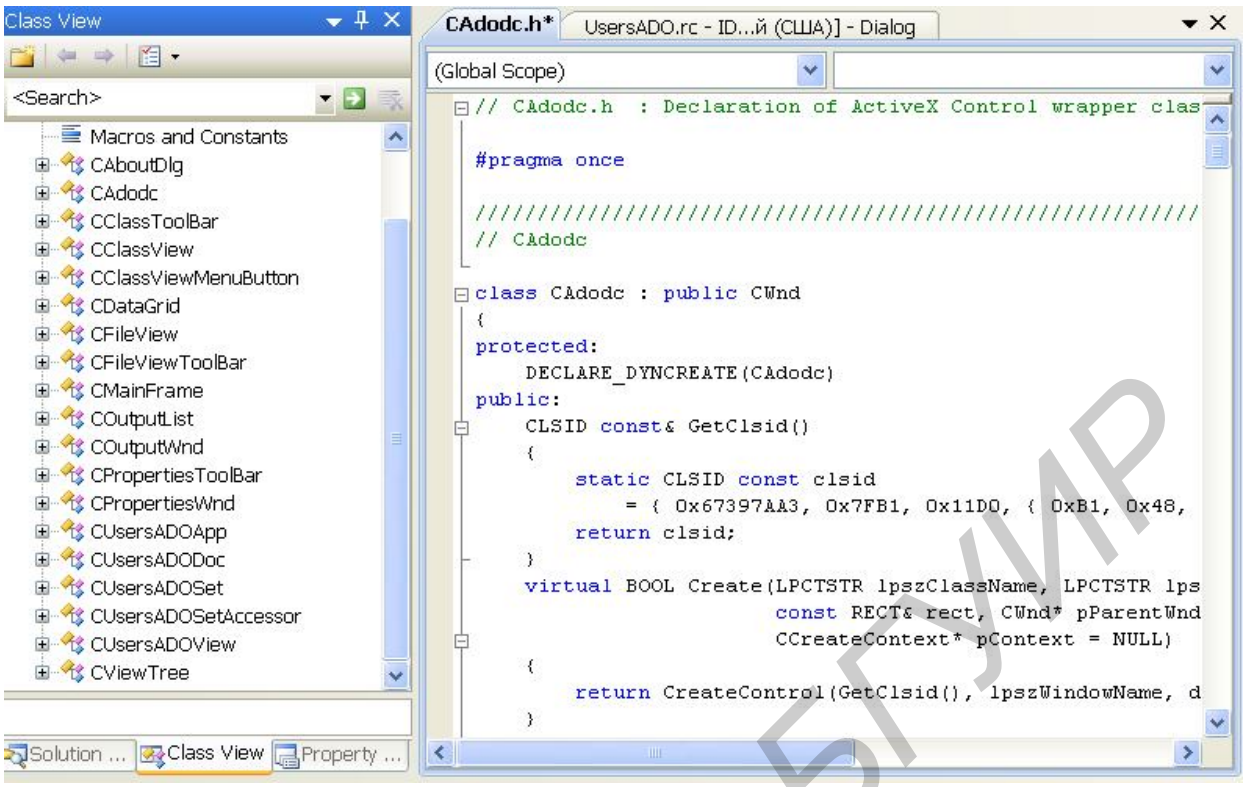

Рис. 19.15. Просмотр классов проекта UsersADO

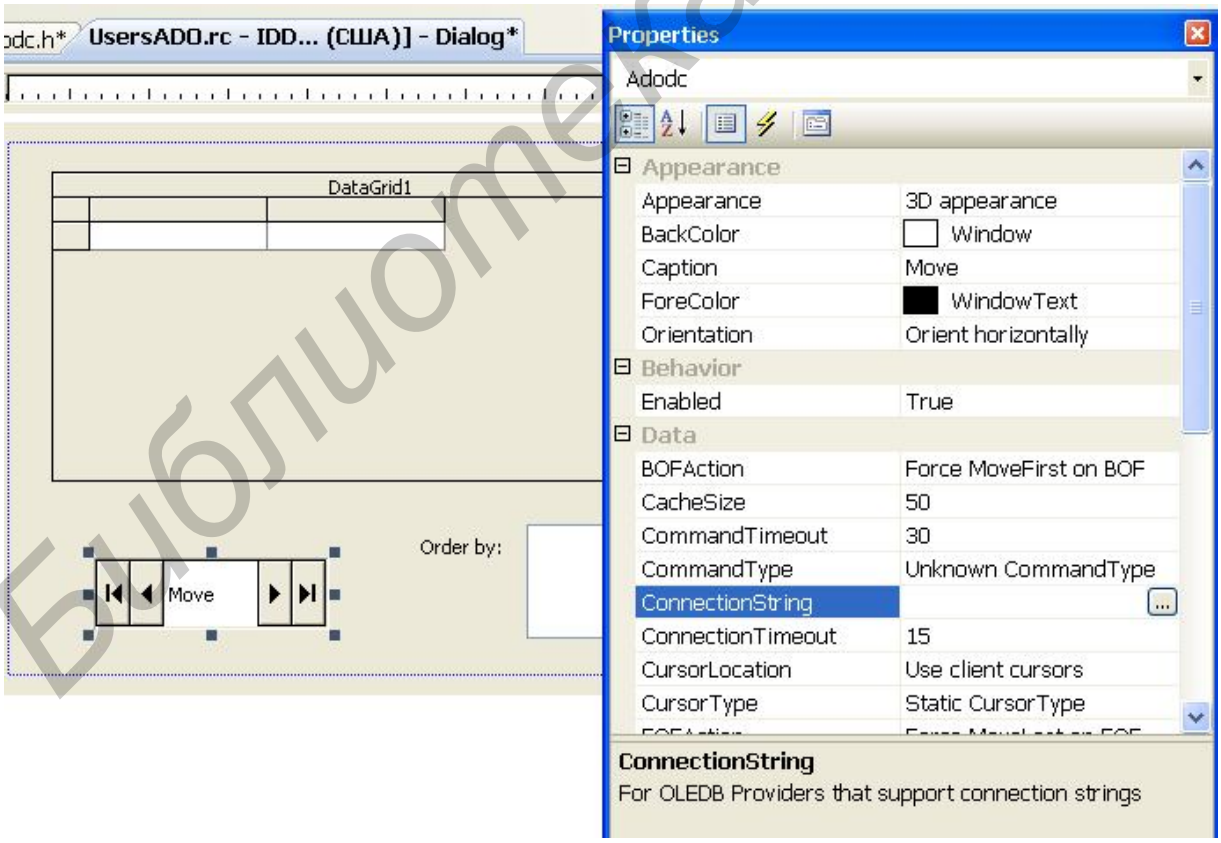

Рис. 19.16. Настройка элемента ADO

19.4. Подключение ADO Data Control к источнику данных

Обратимся к окну Properties элемента ADO. Для изменения заголовка компонента в поле Caption напишем свое название (например Move, рис. 19.16). Выберем свойство Connection String, нажмем на кнопке «…». В появившемся диалоге выберем пункт «Use ODBC Data Source Name», после чего в ставшем активном списке выбираем имя нашего источника данных. Настроим строку ConnectionString (рис. 19.17).

Теперь необходимо сформировать запрос к базе данных. Запрос к базе данных может быть представлен как:

– SQL-запрос к базе данных (1-adCmdText);
– подключение таблицы (2-adCmdTable);
– вызов хранимой процедуры из базы данных (4-adCmdStoredProc).

Выбираем вариант 2-adCmdTable (рис. 19.18).

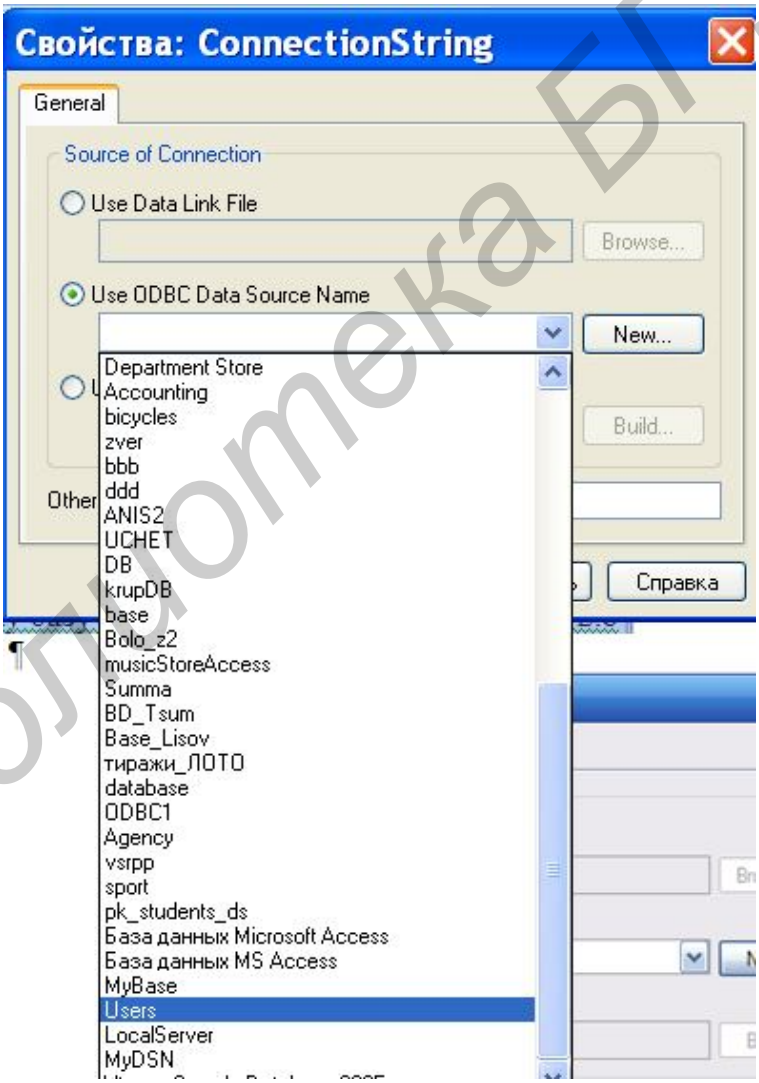

Рис. 19.17. Настройка ConnectionString

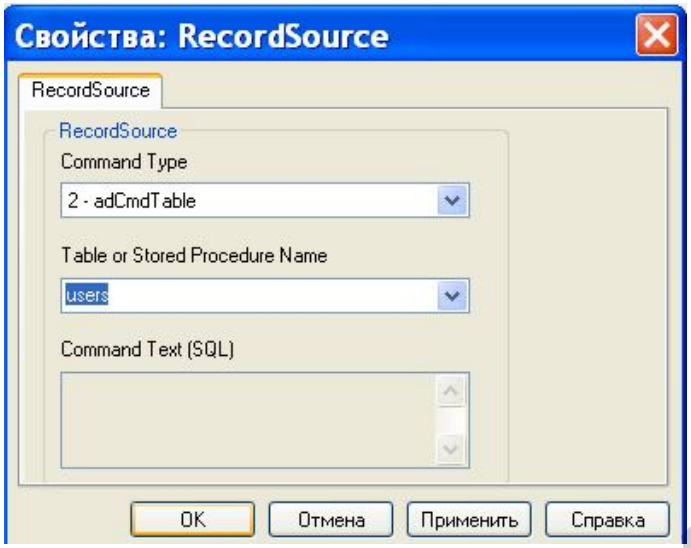

Рис. 19.18. Настройка свойства RecordeSource

19.5. Связывание элементов управления DataGrid Control и ADO Data Control

Отредактируем свойства элемента DataGrid Control с идентификатором IDC_DATAGRID1. Установим свойство DataSource в окне Properties для элемента DataGrid Control равным IDC_ADODC1 (рис. 19.19).

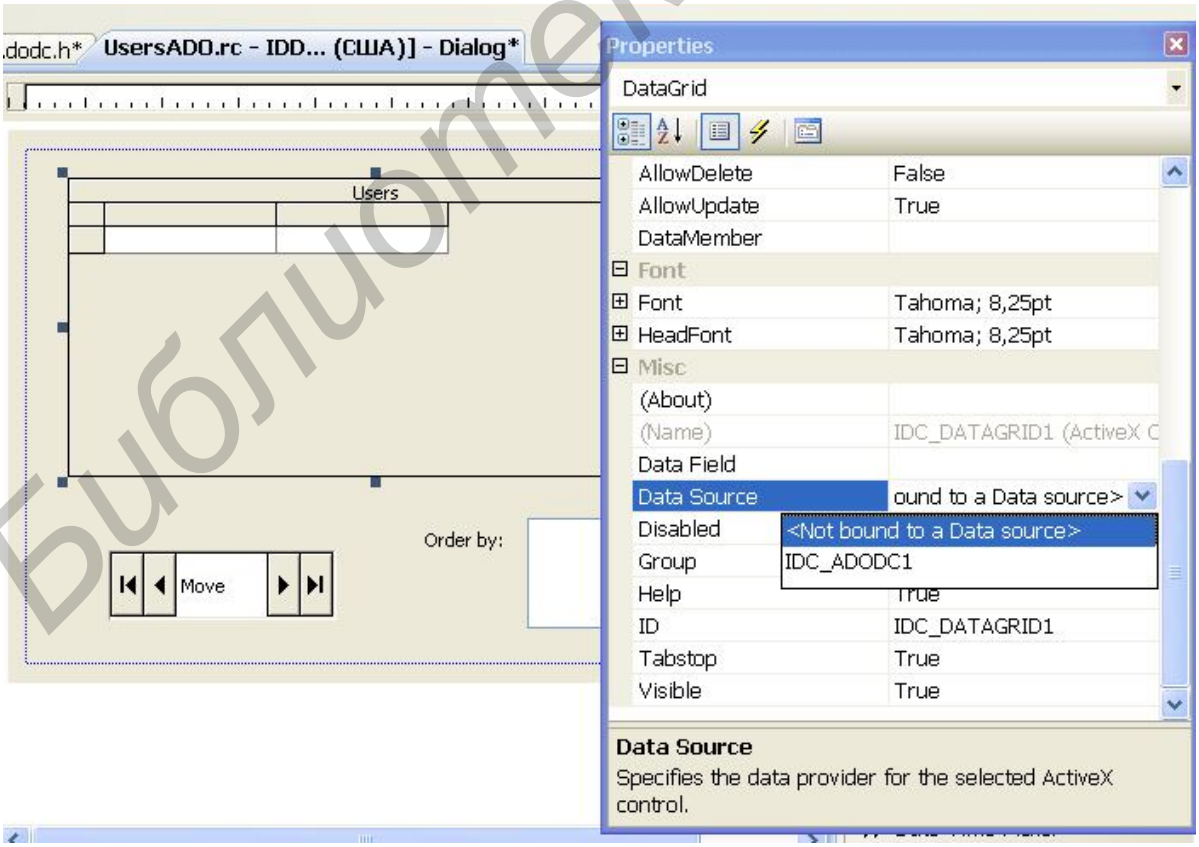

Рис. 19.19. Настройка свойства Data Source элемента DataGrid

Теперь, после компиляции приложения, мы сможем просматривать таблицу базы данных, на которую настроились. Передвигаться по набору записей можно двумя путями:

– пользуясь навигационными кнопками элемента ADO Data Control;

– выбирая записи в элементе DataGrid Control с помощью мыши и клавиш управления курсором.

Если теперь скомпилировать приложение и выполнить, получим отображение на форме текущей выборки таблицы базы данных.

19.6. Удаление, добавление и редактирование записей БД

В текущий момент времени мы можем только лишь просматривать записи из источника данных. Для того чтобы реализовать возможности удаления, добавления и редактирования, необходимо настроить свойства компонента Грид. Выполним такую последовательность действий. Вызовем меню Properties элемента Грид. Перейдем на закладку Control. Поставим пометки True напротив следующих свойств: AllowAddNew, AllowDelete, AllowUpdate (рис. 19.20).

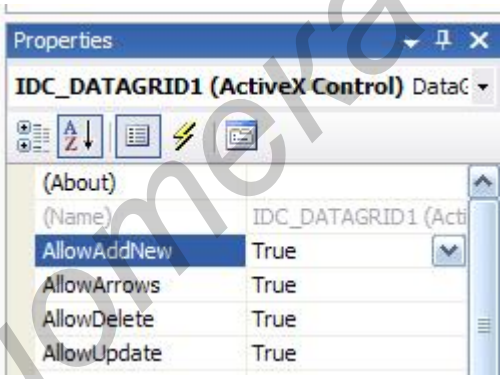

Рис. 19.20. Настройка свойств элемента DataGrid

Настроим свойства элемента Адо. Для этого вызовем меню Properties элемента Адо, в котором выберем закладку All. Теперь установим значения некоторых свойств (рис. 19.21):

– Cursor Locations на 2-Use server cursor;

– Cursor Type на 1-Keyset Cursor Type.

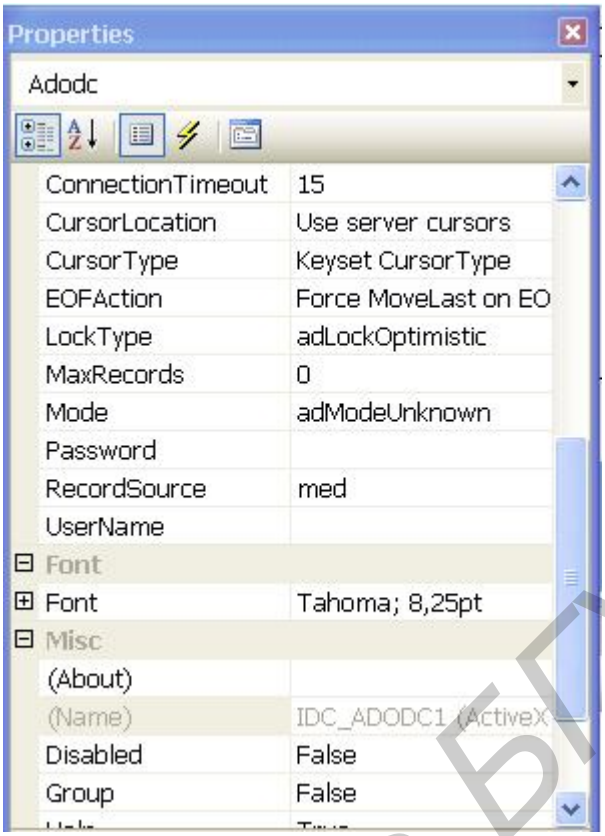

Рис. 19.21. Настройка свойств элемента ADO

После запуска приложения переместимся в ячейку, которую хотим редактировать, и отредактируем ее. Для подтверждения изменений, сделанных в ячейке, необходимо убрать фокус, например нажатием клавиши Enter. Для добавления новой записи необходимо переместиться в самую нижнюю строку элемента Грид и добавить данные. После завершения заполнения этой строки данными и перемещения фокуса на другую строку к элементу Грид автоматически добавляется еще одна строка, которая располагается ниже текущей. Для удаления нужной строки необходимо выделить ее и нажать клавишу Delete.

19.7. Организация сортировки и фильтрации записей

Для организации сортировки и фильтрации текущей выборки базы данных необходимо добавить на форму элементы управления, представленные в табл. 19.1.

Таблица 19.1

Описание элементов управления для организации сортировки и фильтрации

Идентификатор ресурса	**Класс элемента**	**Имя ассоциированной переменной**	**Функциональное назначение**
IDC_ADODC1	CAdodc	m_adodc	Настройка эл-та ADO
IDC_DATAGRID 1	CDataGrid	m_grid	Настройка эл-та Грид
IDC_COMBO1	CComboBox	m_sort	Список выбора поля сортировки
IDC_COMBO2	CComboBox	m_filter_spis	Список выбора поля фильтрации
IDC_CHECK1	CButton	m_napr	Флажок выбора направления сортировки
IDC_EDIT1	CEdit	m_value	Значение фильтра

Для каждого из перечисленных элементов управления добавляются ассоциированные переменные в класс отображения формы (View) (рис. 19.22 и 19.23).

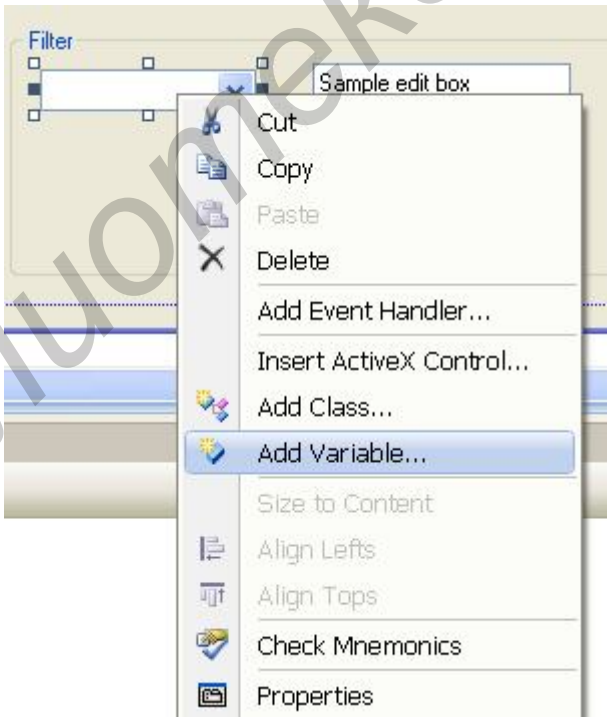

Рис. 19.22. Вызов панели добавления ассоциированной переменной для элемента IDC_COMBO2

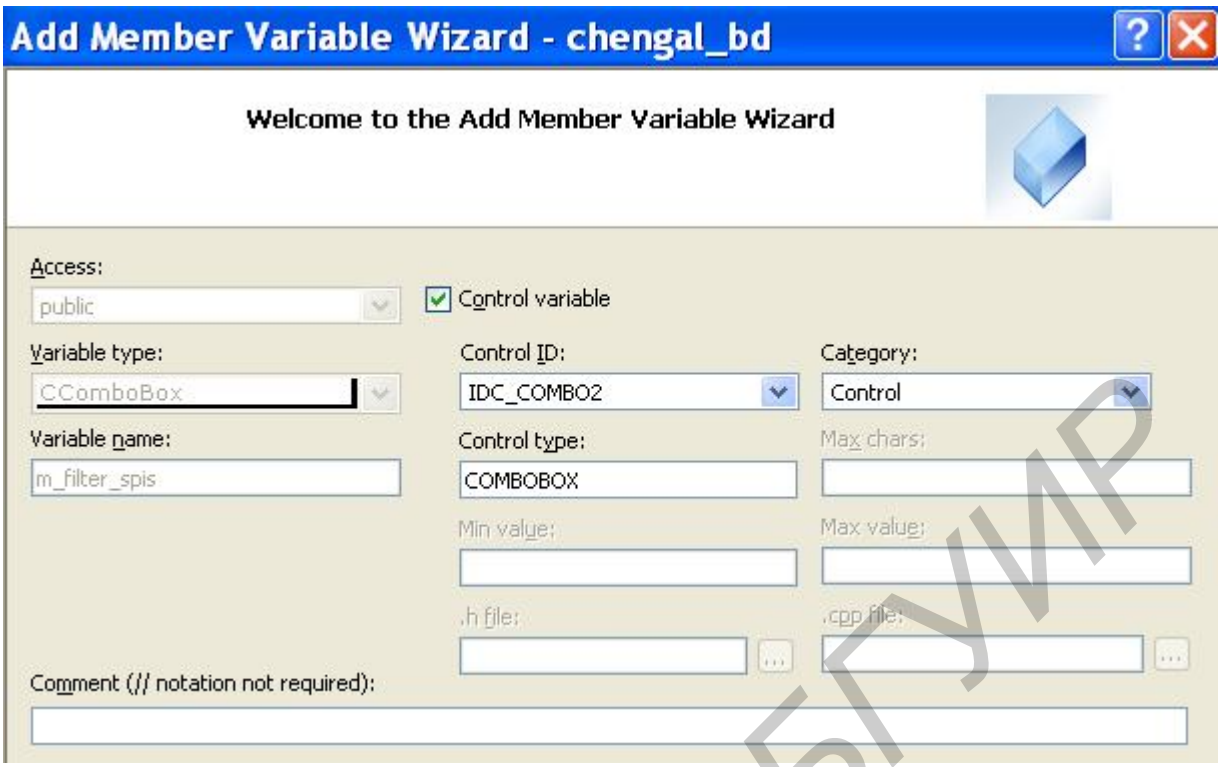

Рис. 19.23. Добавление ассоциированной переменной m_filter_spis

Проинициализировать списки сортировки и фильтрации можно в методе OnInitUpdate() класса представления, чтобы к моменту отображения диалогового окна они содержали информацию. Воспользуемся для этого методом AddString() для объектов m_sort и m_filter_spis:

```
m_sort_value.AddString("unsort");
m_sort_value.AddString("UserID");
m_sort_value.AddString("UserFirstName");
m_sort_value.AddString("UserLastName");
m_sort_value.AddString("UserMiddletName");
m_sort_value.AddString("UserEmail");
m_sort_value.AddString("UserAddress");

m_filter_spis.AddString("unfilter");
m_filter_spis.AddString("UserID");
m_filter_spis.AddString("UserFirstName");
m_filter_spis.AddString("UserLastName");
m_filter_spis.AddString("UserMiddletName ");
m_filter_spis.AddString("UserEmail");
m_filter_spis.AddString("UserAddress");
```

Для выполнения сортировки обработаем событие CBN_SELCHANGE в классе CUsersADOView (рис. 19.24).

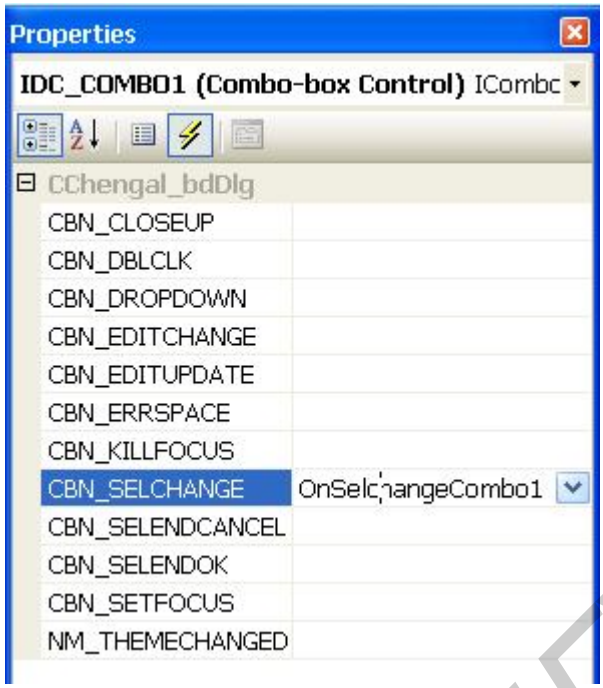

Рис. 19.24. Добавление обработчика события CBN_SELCHANGE

Метод OnSelchangeCombo1() для выполнения сортировки представлен в листинге 19.1.

Листинг 19.1.

```
void CUsersADOView::OnSelchangeCombo1()
{
CString str1, str2;
str1="SELECT * FROM Users";
if(m_sort.GetCurSel()==0) str2="";
else
{
m_sort.GetLBText(m_sort.GetCurSel(),str2);
str2=" ORDER BY " + str2;
if (m_sort_napr.GetCheck()) str2+=" DESC";
}
str1+=str2;
m_adodc.SetRecordSource(str1);
m_adodc.Refresh();
}
```

Принимая выбранное значение из IDC_COMBO1, мы формируем условие отбора в строке SQL-запроса. Выбор в IDC_CHECK1 позволяет сформировать порядок сортировки. Устанавливаем новое свойство элемента m_adodc и получаем из источника данных новую выборку.

Для реализации фильтра добавим на форму кропку IDC_BUTTON1 и создадим обработчик события OnButton1(), как представлено в листинге 19.2.

Листинг 19.2.

```
void CUsersADOView::OnButton1()
{
CString str1, str2,str3;
str1="SELECT * FROM Users ";
m_filter_spis.GetLBText(m_filter_spis.GetCurSel(),str2);
if(str2=="unfilter")
{
  str2="";
  str3="";
}
else
  if(str2!="")
  {
    m_value.GetWindowText(str3);
    str1+="WHERE ";
    str1+=str2;
    str1+=" LIKE ";
    str1+="'%";
    str1+=str3;
    str1+="%'";
  }
m_adodc.SetRecordSource(str1);
m_adodc.Refresh();
}
```

Из IDC_COMBO2 выбирается критерий фильтра, а значение фильтра вводится в поле IDC_EDIT. По нажатии кнопки IDC_BUTTON1 формируется строка запроса, которая устанавливается в свойство RecordSource элемента ADO DC, и текущая выборка обновляется.

Литература

1. Лабораторный практикум по курсу «Визуальные средства разработки приложений» для студентов специальности 40 01 02-02 «Информационные системы и технологии в экономике» / В. Н. Комличенко [и др.]. – Минск : БГУИР, 2002. – 89 с.

2. Визуальные средства разработки приложений : учеб.-метод. пособие по курсу «Объектно-ориентированное проектирование и программирование» для студ. спец. 40 01 02-02 «Информационные системы и технологии в экономике» дневной формы обучения / В. Н. Комличенко [и др.]. – Минск : БГУИР, 2004. – 68 с.

3. Хортон, Айвор. Visual C++ 2005 : Базовый курс / Айвор Хортон ; пер. с англ. – М. : ООО «И. Д. Вильямс», 2007. – 1152 с.

4. Хортон, Айвор. Visual C++ 2010 : Полный курс / Айвор Хортон ; пер. с англ. – М. : ООО «И. Д. Вильямс», 2011. – 1206 с.

5. Грегори, К. Использование Visual C++ 6. Специальное издание / К. Грегори ; пер. с англ. – М. : СПб. ; К. : Издательский дом «Вильямс», 2003. – 864 с.

6. Тихомиров, Ю. Самоучитель MFC / Ю. Тихомиров. – СПб. : БХВ, 2002. – 640 с.

7. Первые шаги [Электронный ресурс]. – Режим доступа: http://www.firststeps.ru/ .

8. Давыдов, В. Г. Visual C++. Разработка Windows-приложений с помощью MFC и API-функций / В. Г. Давыдов. – СПб. : БХВ, 2008. – 576 с.

9. Библиотека MSDN (по-русски) [Электронный ресурс] – Режим доступа: http://msdn.microsoft.com/ru-ru/ .

Св. план 2012, поз. 70

Учебное издание

Кириенко Наталья Алексеевна

РАЗРАБОТКА WINDOWS-ПРИЛОЖЕНИЙ НА ЯЗЫКЕ C++ С ИСПОЛЬЗОВАНИЕМ БИБЛИОТЕКИ MFC

Учебно-методическое пособие

Редактор Н. В. Гриневич
Корректор И. П. Острикова
Компьютерная верстка Ю. Ч. Клочкевич

Подписано в печать. 23.07.2012. Формат 60x84 1/16.Бумага офсетная. Гарнитура «Таймс».
Отпечатано на ризографе. Усл. печ. л. 11,97. Уч.-изд. л. 11,7. Тираж 150 экз. Заказ 136.

Издатель и полиграфическое исполнение: учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники»
ЛИ №02330/0494371 от 16.03.2009. ЛП №02330/0494175 от 03.04.2009.
220013, Минск, П. Бровки, 6