

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

Инженерно-экономический факультет

Кафедра экономической информатики

**Н. А. Кириенко, Е. Н. Живицкая, В. Н. Комличенко**

## **ВИЗУАЛЬНЫЕ СРЕДСТВА РАЗРАБОТКИ ПРОГРАММНЫХ ПРИЛОЖЕНИЙ. ЛАБОРАТОРНЫЙ ПРАКТИКУМ**

*Рекомендовано УМО по образованию в области информатики  
и радиоэлектроники в качестве пособия для направлений специальности  
1-40 05 01-02 «Информационные системы и технологии (в экономике)»  
и 1-40 05 01-10 «Информационные системы и технологии  
(в бизнес-менеджменте)»*

Минск БГУИР 2018

УДК 004.438(076.5)  
ББК 32.973.2я73  
К43

Р е ц е н з е н т ы:

кафедра дискретной математики и алгоритмики  
Белорусского государственного университета  
(протокол №12 от 31.03.2017);

ведущий научный сотрудник государственного научного  
учреждения «Объединенный институт проблем информатики  
Национальной академии наук Беларуси»,  
кандидат технических наук, доцент Н. Н. Парамонов

**Кириенко, Н. А.**

К43 Визуальные средства разработки программных приложений.  
Лабораторный практикум : пособие / Н. А. Кириенко, Е. Н. Живицкая,  
В. Н. Комличенко. – Минск : БГУИР, 2018. – 132 с. : ил.  
ISBN 978-985-543-382-9.

Содержит описания лабораторных работ, выполняемых студентами,  
осваивающими дисциплину «Визуальные средства разработки программных  
приложений». В описании каждой работы представлен краткий теоретический  
материал и подробная инструкция по разработке приложения.

**УДК 004.438(076.5)**  
**ББК 32.973.2я73**

**ISBN 978-985-543-382-9**

© Кириенко Н. А., Живицкая Е. Н.,  
Комличенко В. Н., 2018  
© УО «Белорусский государственный  
университет информатики  
и радиоэлектроники», 2018

## Содержание

Введение .....	4
Лабораторная работа №1. Рисование в окне приложения .....	5
Лабораторная работа №2. Сохранение и восстановление состояния объектов классов .....	28
Лабораторная работа №3. Стандартные элементы управления .....	37
Лабораторная работа №4. Доступ к базам данных с помощью технологии ODBC .....	50
Лабораторная работа №5. Использование технологии OLE DB .....	63
Лабораторная работа №6. Применение технологии ADO для доступа к базе данных .....	77
Лабораторная работа №7. Использование потоков в приложении .....	87
Лабораторная работа №8. Разработка сетевого приложения с использованием Windows sockets .....	100
Приложение 1. Разработка каркаса приложения с помощью мастера AppWizard .....	115
Приложение 2. Диалоговые окна .....	121
Приложение 3. Меню .....	125
Приложение 4. Панель инструментов .....	127
Приложение 5. Генерация приложения, связанного с базой данных .....	129
Список использованных источников .....	131

## ВВЕДЕНИЕ

Дисциплина «Визуальные средства разработки программных приложений» является составной частью цикла дисциплин по информационным технологиям, изучаемых студентами направлений специальности 1-40 05 01-02 «Информационные системы и технологии (в экономике)», 1-40 05 01-10 «Информационные системы и технологии (в бизнес-менеджменте)» на протяжении всего курса обучения в БГУИР.

Целью преподавания данной дисциплины является формирование у студентов базовых понятий и навыков создания программных комплексов в операционной среде Windows, без которых невозможно изучение многих последующих дисциплин данного направления, а также эффективное использование информационных технологий в специальных дисциплинах.

Задачей изучения дисциплины является овладение знаниями и навыками использования языка C++ и среды Microsoft Visual Studio для разработки Windows-приложений, применяемых при автоматизации решения экономических задач.

В процессе изучения дисциплины студенты прослушивают курс лекций и выполняют серию лабораторных работ. В пособии представлено 8 лабораторных работ, даны подробные инструкции по их выполнению, сопровождаемые графическим материалом. Темы лабораторных работ охватывают наиболее важные разделы изучаемой дисциплины. В каждой работе приведен список индивидуальных заданий.

## ЛАБОРАТОРНАЯ РАБОТА №1 РИСОВАНИЕ В ОКНЕ ПРИЛОЖЕНИЯ

Цель работы – научиться создавать простейшее Windows-приложение с использованием библиотеки MFC, изучить архитектуру «документ/представление», реализовать обработку событий классами приложения, разработать программу рисования графических примитивов в окне приложения.

### Методические указания

При разработке программ в VC++ и MFC обычно предполагается использование архитектуры «документ/представление». Названная архитектура позволяет связать данные с их представлением пользователю на экране. Архитектура «документ/представление» предоставляет множество возможностей для работы с документом. Так, *Мастер приложений* 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.

После того как *Мастер приложений* создаст основной шаблон программы с именем MiniDraw (прил. 1), в класс представления необходимо добавить код для отслеживания действий мыши и рисования прямых линий в окне представления. Для создания обработчиков сообщений мыши и настройки окна представления используется *Мастер классов*, для изменения меню программы – редактор ресурсов.

В первую очередь необходимо определить класс для сохранения данных о каждой созданной линии. В начало файла заголовка MiniDrawDoc.h перед определением класса CMiniDrawDoc добавьте следующее определение класса CLine:

```
class CLine : public CObject
{
protected :
    int m_X1, m_Y1, m_X2 , m_Y2;
```

```

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

```

Переменные `m_X1` и `m_Y1` класса `CLine` сохраняют координаты одного конца прямой линии, `m_X2` и `m_Y2` – другого. Класс `CLine` содержит также функцию `Draw` для рисования линии.

В эту функцию передается указатель на объект класса `CDC`. Он необходим для вызова рисующих функций контекста устройства. Класс `CDC` инкапсулирует функции инициализации контекста устройства, а также функции рисования.

Теперь добавим в класс `CMiniDrawDoc` требуемые члены, вводя в начало определения класса следующие операторы:

```

class CMiniDrawDoc : public CDocument
{
protected :
    CTypedPtrArray<CObArray , CLine*> m_LineArray;
public :
    void AddLine (int X1, int Y1, int X2, int Y2) ;
    CLine *GetLine (int Index) ;
    int GetNumLines () ;
    // остальные определения класса CMiniDrawDoc...
};

```

*Примечание.* В файл `MiniDrawDoc.h` надо подключить файл `#include "afxtempl.h"`

Новая переменная `m_LineArray` – это экземпляр шаблона `CTypedPtrArray`. Класс `CTypedPtrArray` генерирует семейство классов, каждый из которых является производным от класса, заданного в первом параметре шаблона (им может быть `CObArray` или `CPtrArray`). Каждый из этих классов предназначен для хранения переменных класса, описанных вторым параметром шаблона. Таким образом, переменная `m_LineArray` является объектом класса, порожденного от класса `CObArray` и сохраняющего указатели на объекты класса `CLine`.

Класс `CObArray` – это один из классов коллекций общего назначения в библиотеке MFC, используемый для сохранения групп переменных или объектов. Экземпляр класса `CObArray` хранит множество указателей на объекты класса `CObject` (или любого класса, порожденного от `CObject`) в структуре данных, подобной массиву. `CObject` – это MFC-класс, от которого прямо или косвенно порождаются практически все остальные классы. Однако вместо использования экземпляра класса общего назначения

CObArray программа MiniDraw использует шаблон CTypedPtrArray, спроектированный специально для хранения объектов класса CLine. Это позволяет компилятору выполнять более интенсивный контроль соответствия типов данных, уменьшать число ошибок и сокращать число операций приведения типов при использовании объектов класса.

Для использования шаблона класса CTypedPtrArray в стандартный файл заголовков StdAfx.h нужно включить Afxtempl.h:

```
#include<afxwin.h> // стандартные компоненты MFC
#include<afxext.h> // расширения библиотеки MFC
#include<afxdtctl.h> // поддержка общих элементов управления
#include <afxtempl.h> // шаблоны библиотеки MFC
#ifdef _AFX_NO_AFXCMN_SUPPORT
#include <afxcmn.h> // поддержка общих элементов
#endif // _AFX_NO_AFXCMN_SUPPORT
```

В MiniDrawDoc переменная m\_LineArray используется для хранения указателя на каждый объект класса CLine, сохраняющий информацию о линии. Функции класса AddLine, GetLine и GetNumLines предоставляют доступ к информации о линии, хранящейся в массиве m\_LineArray (другие классы не имеют к нему прямого доступа, т. к. переменная m\_LineArray является защищенной). Теперь в конце файла реализации документа MiniDrawDoc.cpp введите определение (код) функции CLine::Draw:

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

Чтобы создать линию по координатам, сохраненным в текущем объекте, функция Draw вызывает две функции класса CDC – MoveTo и LineTo. Далее в конце файла MiniDrawDoc.cpp добавьте определения функций AddLine, GetLine и GetNumLines класса CminiDrawDoc:

```
void CMiniDrawDoc::AddLine(int X1, int Y1, int X2, int Y2)
{
    CLine *pLine=new CLine(X1, Y1, X2, Y2);
    m_LineArray.Add(pLine);
}
CLine* CMiniDrawDoc::GetLine(int Index)
{
    if(Index<0||Index>m_LineArray.GetUpperBound ())
        return 0;
    return m_LineArray.GetAt(Index);
}
int CMiniDrawDoc::GetNumLines()
{
```

```

return (int)m_LineArray.GetSize();
}

```

Чтобы добавить указатель объекта в коллекцию указателей на класс CLine, сохраненных в массиве m\_LineArray, функция AddLine создает новый объект класса CLine и вызывает функцию Add класса CObArray.

Указатели, сохраненные в массиве m\_LineArray, индексируются. Первый добавленный указатель имеет индекс 0, второй 1 и т. д. Функция GetLine возвращает указатель с индексом, содержащимся в переданном параметре. Вначале она контролирует попадание индекса в допустимый интервал значений. Функция GetUpperBound класса CObArray возвращает наибольший допустимый индекс, т. е. индекс последнего добавленного указателя. Далее функция GetLine возвращает соответствующий указатель класса CLine, получаемый в результате вызова функции GetAt класса CTypedPtrArray. Функция GetNumLines возвращает количество объектов класса CLine, сохраненных в переменной m\_LineArray. Для этого вызывается функция GetSize класса CObArray. Функции AddLine, GetLine и GetNumLines вызываются функциями – членами класса представления.

Добавим в класс представления несколько переменных: m\_className, m\_Dragging, m\_HCross, m\_PointOld и m\_PointOrigin. Для этого откройте файл MiniDrawView.h и добавьте выражения, выделенные курсивом, в начало определения класса CMiniDrawView:

```

class CMiniDrawView : public CView
{
protected:
    CString m_ClassName;
    int m_Dragging;
    HCURSOR m_HCross;
    CPoint m_PointOld;
    CPoint m_PointOrigin;
};

```

Добавьте в конструктор класса CMiniDrawView в файле реализации код инициализации переменных m\_Dragging и m\_HCross:

```

CMiniDrawView::CMiniDrawView()
{
    m_Dragging = 0;
    m_HCross=AfxGetApp()->LoadStandardCursor(IDC_CROSS);
}

```

Переменная m\_HCross хранит дескриптор указателя мыши, отображаемого программой, когда тот находится внутри окна представления. Функция AfxGetApp возвращает указатель на объект класса приложения (класс CMiniDrawApp, порожденный от класса CWinApp), где он используется для вызова функции LoadStandardCursor класса CWinApp. Эта функция при получении

идентификатора IDC\_CROSS возвращает дескриптор стандартного крестообразного указателя. В табл. 1.1 приведены значения, которые можно передать в функцию LoadStandardCursor для получения дескрипторов других стандартных указателей.

Таблица 1.1

Значения дескрипторов стандартных указателей

IDC_ARROW	Стандартный указатель-стрелка
IDC_CROSS	Перекрестье, используемое для выбора
IDC_IBEAM	Указатель для редактирования текста
IDC_SIZEALL	Указатель из четырех стрелок для изменения размеров окна
IDC_SIZENESW	Двунаправленная стрелка, указывающая на северо-восток и юго-запад
IDC_SIZENS	Двунаправленная стрелка, указывающая на север и юг
IDC_SIZENWSE	Двунаправленная стрелка, указывающая на северо-запад и юго-восток
IDC_SIZEWE	Двунаправленная стрелка, указывающая на запад и восток
IDC_UPARROW	Вертикальная стрелка
IDC_WAIT	«Песочные часы», используемые при длительном выполнении задачи

AfxGetApp – это глобальная функция библиотеки MFC, которая не является членом класса и содержит глобальные функции, начинающиеся с префикса Afx.

Чтобы пользователь мог рисовать линии внутри окна представления с помощью мыши, программа должна реагировать на события, происходящие внутри этого окна. Для обработки сообщения, передаваемого мышью, в класс представления необходимо добавить функцию обработки сообщений.

**Обработка сообщений.** С каждым окном в графической программе связана функция, называемая процедурой окна. Когда происходит некоторое событие, операционная система вызывает эту функцию, передавая ей идентификатор происшедшего события и данные для его обработки. Подобный процесс называется передачей сообщения окну.

Процедура окна с помощью библиотеки MFC создается автоматически. Если необходима собственная обработка сообщения, то создается функция обработки сообщения, являющаяся членом класса управления окном. Для определения обработчика сообщения можно воспользоваться кнопкой **Сообщения** в закладке **Свойства** для нужного класса, как описано ниже.

Например, если указатель находится внутри окна представления, то при щелчке левой кнопки мыши передается идентификатор WM\_LBUTTONDOWN. Чтобы предусмотреть собственную обработку этого сообщения, нажмите на нужный класс, вызовите всплывающее меню, в котором выберите команду **Свойства**, как показано на рис. 1.1.

После чего на экране появится закладка **Свойства** в правой части окна, как показано на рис. 1.2. В закладке **Свойства** нажмите кнопку **Сообщения**, после чего появится окно, представленное на рис. 1.3. В нем выберите необходимый



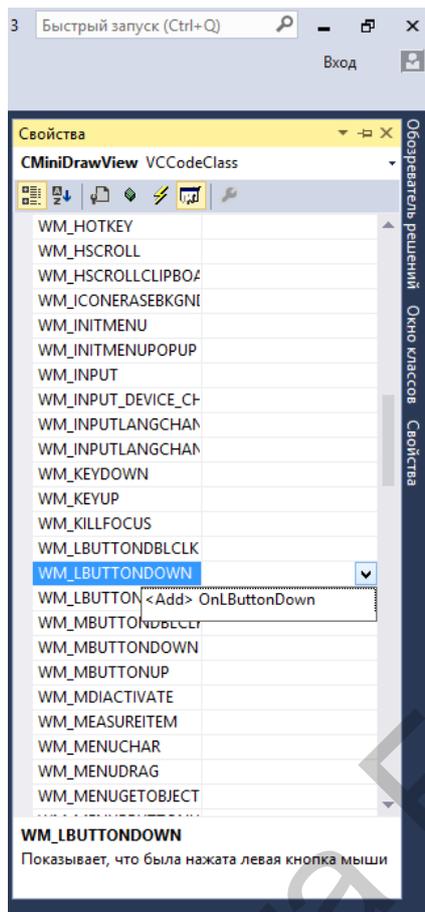


Рис. 1.3. Окно добавления обработчика для сообщения WM\_LBUTTONDOWN

**Командные сообщения.** Сообщения, генерируемые объектами пользовательского интерфейса, называют командными сообщениями. Каждый раз, когда пользователь выбирает объект интерфейса или когда один из этих объектов необходимо обновить, объект передает командное сообщение главному окну. Однако библиотека MFC сразу направляет сообщение объекту окна представления. Если он не имеет нужного обработчика, библиотека MFC направляет сообщение объекту документа. Если же объект документа не содержит обработчик, библиотека MFC направляет сообщение объекту главного окна программы. Если главное окно также не располагает обработчиком, сообщение направляется объекту приложения. Наконец, если объект приложения не обеспечивает обработку, то сообщение обрабатывается стандартным образом.

Таким образом, библиотека MFC расширяет основной механизм сообщений, чтобы командные сообщения обрабатывались не только объектами, управляющими ими, но и любыми другими объектами приложения. Каждый из них принадлежит классу, прямо или косвенно порожденному от класса `CCommandTarget`, реализующего механизм передачи сообщений.

Важной особенностью такого механизма является то, что программа может обрабатывать нужное сообщение внутри наиболее подходящего для

этого класса. Например, в программе, созданной мастером Application Wizard, команда Exit в меню File обрабатывается классом приложения, т. к. эта команда воздействует на приложение в целом. С другой стороны, команда Save в меню File обрабатывается классом документа, т. к. этот класс отвечает за хранение и запись данных документа.

**Функция OnLButtonDown.** Следующая задача состоит в определении обработчика сообщения WM\_LBUTTONDOWN, которое передается при каждом щелчке левой кнопки мыши, если указатель находится внутри окна представления. Для определения функции выполните следующие действия:

1. Выберите закладку **Окно классов**.
2. Найдите класс CMiniDrawView и вызовите всплывающее меню, в котором выберите команду **Свойства**, как показано на рис. 1.1. После чего на экране появится окно **Свойства**, как показано на рис. 1.2.
3. В окне **Свойства** выберите закладку **Сообщения**, как показано на рис. 1.3. Выбор **Сообщения** задает функцию-член для обработки любого уведомляющего сообщения, переданного окну представления, что позволяет переопределить одну из виртуальных функций, которую класс CMiniDrawView наследует от CView и других базовых классов библиотеки MFC. Другие закладки (**События**, **Переопределения** и т. д.) в окне **Свойства** определяют сообщения, поступающие от объектов интерфейса (команды меню).
4. В списке Сообщений выберите идентификатор WM\_LBUTTONDOWN, обрабатываемый определяемой функцией. Закладка Сообщения содержит идентификаторы всех уведомляющих сообщений, которые передаются окну представления (идентификаторы сообщений – это константы, записанные заглавными буквами и начинающиеся с префикса WM\_). При выборе конкретного идентификатора сообщения внизу окна Свойства появляется соответствующее краткое описание.
5. Справа от идентификатора обработчика во всплывающем списке выберите команду <Add> OnLButtonDown, как показано на рис. 1.3. Выбор команды объявляет функцию в определении класса CMiniDrawView в файле MiniDrawView.h, вносит ее определение в файл MiniDrawView.cpp и добавляет функцию в схему сообщений класса. Теперь справа от имени сообщения в списке Messages отображается заданный обработчик.
6. Справа от идентификатора обработчика во всплывающем списке выберите команду <Edit Code>, после чего откроется файл MiniDrawView.cpp (если он еще не открыт) и отобразит только что созданную функцию OnLButtonDown, чтобы можно было добавить ее код.
7. Добавьте в функцию OnLButtonDown операторы, выделенные курсивом:

```
void CMiniDrawView::OnLButtonDown(UINT nFlags, CPoint point)
{
    m_PointOld=point;
    m_PointOrigin=point;
```

```

SetCapture();
m_Dragging=1;
RECT Rect;
GetClientRect(&Rect);
ClientToScreen(&Rect);
::ClipCursor(&Rect);
CView::OnLButtonDown(nFlags, point);
}

```

Если указатель находится в окне представления, то после щелчка левой кнопкой мыши управление будет передано функции `OnLButtonDown`, а параметр *point* будет содержать текущую позицию указателя. Эта позиция сохраняется в переменных `m_PointOrigin` и `m_PointOld`. Переменная `m_PointOrigin` хранит координаты начальной точки линии. Переменная `m_PointOld`, как вы скоро увидите, используется другими обработчиками сообщений для получения информации о положении указателя мыши в момент предыдущего сообщения.

Мастер добавляет строку в функцию `OnLButtonDown`, вызывающую функцию `OnLButtonDown`, определенную в базовом классе. Это делается для того, чтобы базовый класс мог выполнять стандартную обработку сообщений.

Вызов функции `SetCapture` класса `CWnd` приводит к захвату мыши, и все последующие ее сообщения передаются в окно представления, пока захват не будет отменен. Таким образом, окно представления полностью контролирует мышь в процессе рисования линии. Значение переменной `m_Dragging` устанавливается равным 1, что информирует других обработчиков сообщений о выполнении операции рисования.

Оставшийся фрагмент программы предназначен для ограничения перемещения указателя мыши границами окна представления. Функция `CWnd::GetClientRect` возвращает текущие координаты окна представления, а `CWnd::ClientToScreen` преобразовывает их в экранные (т. е. координаты, заданные по отношению к верхнему левому углу экрана). Наконец, функция `::ClipCursor` ограничивает перемещения указателя мыши в пределах заданных координат, удерживая его в окне представления.

Функция `::ClipCursor` содержится в Win32 API, а не в MFC. Поскольку она описана как глобальная, ее имени предшествует операция расширения области видимости (`::`). Использование данной операции не является необходимым, если глобальная функция не скрыта функцией-членом с таким же именем.

**Схема сообщений.** Когда мастер создает обработчик сообщения, то помимо объявления и определения функции-члена он также добавляет ее в специальную структуру MFC, называемую схемой (картой) сообщений (`message map`) и связывающую функции с обрабатываемыми сообщениями. Схема сообщений позволяет библиотеке MFC вызывать для каждого типа сообщения соответствующий обработчик.

**Мастер приложений** создает необходимый код для реализации схемы сообщений, основанной на наборе MFC-макросов. После применения **Мастера классов** для определения обработчиков трех видов сообщений мыши (создание которых рассматривается ниже) в файл CMiniDrawView.h будут добавлены следующие макросы и объявления функций:

```
protected:
    DECLARE_MESSAGE_MAP()
public:
    afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
public:
    afx_msg void OnMouseMove(UINT nFlags, CPoint point);
public:
    afx_msg void OnLButtonUp(UINT nFlags, CPoint point);
```

В файл реализации представления MiniDrawView.cpp будут добавлены соответственно следующие макросы:

```
BEGIN_MESSAGE_MAP(CMiniDrawView, CView)
ON_WM_MOUSEMOVE()
ON_WM_LBUTTONDOWN()
ON_WM_LBUTTONUP()
END_MESSAGE_MAP()
```

Когда сообщение передается объекту класса, MFC обращается к схеме сообщений, чтобы определить, есть ли в классе обработчик такого сообщения. Если обработчик найден, ему передается управление. При отсутствии обработчика MFC ищет его в базовом классе. Если это не дает результата, то поиск будет продолжен по иерархии классов до первого встретившегося обработчика. Если в иерархии обработчик отсутствует, то будет выполнена стандартная обработка сообщения. Если же это командное сообщение, то оно перенаправляется следующему объекту в описанной ранее последовательности.

Некоторые из классов библиотеки MFC, от которых порождены классы программы, содержат обработчики сообщений. Следовательно, даже если производный класс не определяет обработчик сообщения, обработчик базового класса может обеспечить соответствующую обработку. Например, базовый класс документа CDocument содержит обработчики сообщений, поступающих при выборе команд **Сохранить**, **Сохранить как...** и **Выйти** в меню **Файл** (соответственно OnFileSave, OnFileSaveAs и OnFileClose).

**Функция OnMouseMove.** Далее следует определить функцию для обработки сообщения WM\_MOUSEMOVE. Так как пользователь перемещает указатель мыши внутри окна представления, это окно получает последовательность сообщений WM\_MOUSEMOVE, каждое из которых содержит информацию о текущей позиции указателя. Для определения обработчика таких сообщений выполните последовательность действий,

описанную для функции OnLButtonDown. Однако на шаге 4 в списке **Сообщений** выберите сообщение WM\_MOUSEMOVE вместо сообщения WM\_LBUTTONDOWN.

После выбора команды **<Edit Code>** во всплывающем меню добавьте в функцию OnMouseMove следующие строки:

```
void CMiniDrawView::OnMouseMove(UINT nFlags, CPoint point)
{
    ::SetCursor (m_HCross);
    if (m_Dragging)
    {
        CClientDC ClientDC (this);
        ClientDC.SetROP2(R2_NOT);
        ClientDC.MoveTo(m_PointOrigin);
        ClientDC.LineTo(m_PointOld);
        ClientDC.MoveTo (m_PointOrigin);
        ClientDC.LineTo(point);
        m_PointOld=point;
    }
    CView::OnMouseMove(nFlags, point);
}
```

При перемещении указателя мыши внутри окна представления функция OnMouseMove вызывается через определенные промежутки времени. Добавленный в нее код обеспечивает решение двух основных задач: первой – вызов API-функции ::SetCursor для отображения крестообразного курсора вместо стандартного курсора-стрелки, второй – выполнение операции рисования (значение переменной m\_Dragging отлично от нуля). При этом выполняются следующие действия:

1. Стирание линии, нарисованной при получении предыдущего сообщения WM\_MOUSEMOVE (если она имеется).

2. Рисование новой линии от начальной точки, в которой была нажата левая кнопка мыши с координатами, сохраняемыми в переменной m\_PointOrigin, до текущей позиции указателя, заданной параметром point.

3. Сохранение текущей позиции указателя в переменной m\_PointOld.

Для рисования внутри окна функция OnMouseMove создает объект контекста устройства, связанный с окном представления. Затем OnMouseMove вызывает функцию CDC::SetROP2, задающую режим рисования, в котором линии строятся методом инвертирования (обращения) текущего цвета экрана. В этом режиме линия, нарисованная в определенной позиции в первый раз, будет видима, а при повторном выводе в той же самой позиции – невидима. Таким образом, обработчики сообщения легко отображают и удаляют группы временных линий. Линии выводятся с помощью CDC::MoveTo, указывающей положение одного конца линии, и CDC::LineTo, задающей положение другого конца.

Окончательный результат использования `OnMouseMove` состоит в том, что при перемещении указателя с нажатой кнопкой мыши внутри окна представления временная линия всегда соединяет свое начало с текущей позицией указателя, показывает, какой будет постоянная линия, если пользователь отпустит кнопку мыши сейчас.

**Функция `OnLButtonUp`.** Наконец, необходимо определить функцию для обработки сообщения `WM_LBUTTONDOWN`, передаваемого при отпускании левой кнопки мыши. Чтобы создать функцию, используйте ту же последовательность действий, как и для двух предыдущих сообщений. Однако в списке `Messages` диалогового окна мастера выберите идентификатор `WM_LBUTTONDOWN`. Когда функция будет сгенерирована – добавьте в ее определение в файле `MiniDraw.cpp` следующий текст:

```
void CMiniDrawView::OnLButtonUp(UINT nFlags, CPoint point)
{
    if (m_Dragging)
    {
        m_Dragging = 0;
        ::ReleaseCapture();
        ::ClipCursor (NULL);
        CClientDC ClientDC (this);
        ClientDC.SetROP2 (R2_NOT);
        ClientDC.MoveTo (m_PointOrigin);
        ClientDC.LineTo (m_PointOld);
        ClientDC.SetROP2 (R2_COPYPEN);
        ClientDC.MoveTo (m_PointOrigin);
        ClientDC.LineTo (point);
        CMiniDrawDoc* pDoc=GetDocument();
        pDoc->AddLine(m_PointOrigin.x, m_PointOrigin.y, point.x, point.y);
    }
    CView::OnLButtonUp(nFlags, point);
}
```

Если пользователь перемещает указатель мыши с нажатой кнопкой (значение переменной `m_Dragging` отлично от нуля), добавленный код завершает операцию рисования и строит постоянную линию. В частности, выполняются следующие действия:

- присваивание значения «0» переменной `m_Dragging`, что информирует других обработчиков сообщений о завершении операции рисования;

- вызов API-функции `::ReleaseCapture` для завершения захвата мыши. После этого сообщения мыши будут снова передаваться любому окну, в котором находится указатель;

- передача указателя `NULL` в API-функцию `::ClipCursor`, что позволяет пользователю снова перемещать указатель мыши по всему экрану;

- стирание временной линии, выведенной предыдущим обработчиком сообщения WM\_MOUSEMOVE;
- вывод постоянной линии от начальной точки до текущей позиции указателя;
- добавление новой линии в список существующих линий.

WM\_LBUTTONDOWN – это последнее сообщение мыши, которое нужно обработать в программе MiniDraw. В табл. 1.2 приведен список уведомляющих сообщений.

Таблица 1.2

Список уведомляющих сообщений

WM_MOUSEMOVE	Указатель мыши перемещен на новое место внутри рабочей области
WM_LBUTTONDOWN	Нажата левая кнопка
WM_MBUTTONDOWN	Нажата средняя кнопка
WM_RBUTTONDOWN	Нажата правая кнопка
WM_LBUTTONUP	Отпущена левая кнопка
WM_MBUTTONUP	Отпущена средняя кнопка
WM_RBUTTONUP	Отпущена правая кнопка
WM_LBUTTONDOWNDBCLICK	Выполнено двойное нажатие левой кнопки
WM_MBUTTONDOWNDBCLICK	Выполнено двойное нажатие средней кнопки
WM_RBUTTONDOWNDBCLICK	Выполнено двойное нажатие правой кнопки

**Параметры сообщений мыши.** Всем обработчикам сообщений передаются два параметра: nFlags и point.

Параметр nFlags показывает состояние кнопок мыши и некоторых клавиш в момент наступления события. Состояние каждой кнопки или клавиши представляется специальным битом. Для обращения к отдельным битам можно использовать битовые маски (табл. 1.3). Например, в следующем фрагменте проверяется, была ли нажата клавиша Shift при перемещении мыши:

```
void CMiniDrawView::OnMouseMove(UINT nFlags, CPoint point)
{
    if (nFlags&MK_SHIFT)
        // клавиша Shift была нажата при перемещении мыши
}
```

Параметр point – это структура CPoint, задающая координаты курсора мыши в тот момент, когда произошло событие мыши. Поле x (point.x) содержит горизонтальную координату указателя, поле y (point.y) – вертикальную. Координаты определяют местоположение указателя относительно верхнего левого угла окна. Точнее говоря, параметр point задает координаты острия указателя мыши. Острие – это отдельный пиксель внутри указателя, выделяемый при его проектировании. Острием стандартного курсора-стрелки является ее конец, а острием стандартного

крестообразного курсора – точка пересечения его линий. В табл. 1.3 приведены битовые маски для доступа к битам параметра nFlags, передаваемого обработчику сообщения мыши.

Таблица 1.3

Битовые маски для доступа к битам параметра nFlags

MK_CONTROL	Ctrl
MK_LBUTTON	Левая кнопка мыши
MK_MBUTTON	Средняя кнопка мыши
MK_RBUTTON	Правая кнопка мыши
MK_SHIFT	Shift

**Перерисовка окна.** Теперь программа постоянно хранит данные, позволяющие восстановить линию, а класс представления может использовать их при перерисовке окна. Вспомните: для перерисовки окна система удаляет его содержимое, а затем вызывает функцию OnDraw класса представления. В минимальную версию функции OnDraw, генерируемую мастером Application Wizard, необходимо добавить собственный код для перерисовки окна. Для этого в функцию CMiniDrawView::OnDraw в файле MiniDrawView.cpp необходимо добавить строки, выделенные курсивом:

```
void CMiniDrawView::OnDraw(CDC* pDC)
{
    CMiniDrawDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    if (!pDoc)
        return;
    int Index=pDoc->GetNumLines();
    while(Index--)
        pDoc->GetLine (Index)->Draw (pDC);
}
```

В этом коде вызывается функция CMiniDrawDoc::GetNumLines, позволяющая определить количество линий, сохраненных объектом документа. В этом фрагменте программы для каждой линии сначала вызывается функция CMiniDrawDoc::GetLine, которая получает указатель на соответствующий объект класса CLine, а затем этот указатель используется для рисования линии с помощью CLine::Draw.

Так как графические данные хранятся внутри объекта документа, целесообразно добавить в программу некоторые команды меню **Правка**, чтобы можно было эти данные изменять. Далее будут добавлены команды **Отменить** для удаления последней нарисованной линии и **Удалить всё** для удаления всех линий.

Рассмотрим, как можно использовать редактор ресурсов для настройки программы MiniDraw (меню и значка).

В окне *Обозреватель решений* откройте вкладку *Окно ресурсов* и разверните граф ресурсов программы всех категорий: Accelerator, Dialog, Icon, Menu, String Table и Version.

Чтобы настроить меню программы, сделайте следующее. Выберите закладку *Окно ресурсов*. Выполните двойной щелчок на идентификаторе IDR\_MAIN\_FRAME в разделе Menu. Обратите внимание: идентификатор IDR\_MAIN\_FRAME используется также для таблицы горячих клавиш и главного значка программы. Developer Studio открывает окно редактора меню, отображающее меню программы MiniDraw, созданное *Мастером приложений*. Стандартное меню приложения, созданное по умолчанию, содержит меню *Файл, Правка, Вид, Справка* и др.

Выполните одиночный щелчок на пустом прямоугольнике внутри меню *Правка* (под заголовком) и введите *&Назад*. Теперь в меню *Правка* появится команда *Назад*. В окне *Свойства* в поле *ID* введите ID\_EDIT\_UNDO.

Выполните одиночный щелчок на пустом поле внизу меню *Правка* (под командой *Назад*) и введите Separator. После чего в окне *Свойства* для поля Separator выберите значение true. Под командой *Назад* будет вставлен разделитель.

Выполните одиночный щелчок на пустом поле внизу меню *Правка* и введите *&Удалить всё*, затем в окне *Свойства* в поле *ID* введите значение ID\_EDIT\_CLEAR\_ALL. В меню добавится команда *Удалить всё*. Теперь меню *Правка* завершено. Закройте окно редактора меню и сохраните результаты, выбрав команду *Сохранить всё* в меню *Файл* или щелкнув на кнопке *Сохранить всё* панели инструментов Standard.

Основная информация о ресурсах хранится в файле определения ресурсов MiniDraw.rc, а информация о значке – в MiniDraw.ico подкаталога \res каталога проекта. Файл определения ресурсов содержит оператор ICON, идентифицирующий файл значка. Когда программа будет сгенерирована, программа Rc.exe (Resource Compiler – компилятор ресурсов) обработает информацию о ресурсах, содержащуюся в этих файлах, и внесет данные о них в исполняемый файл.

**Настройка окна MiniDraw.** При использовании программы MiniDraw в настоящем виде возникают две проблемы.

Первая: хотя обработчик сообщения WM\_MOUSEMOVE отображает требуемый указатель крестообразной формы, Windows также пытается отобразить стандартный курсор-стрелку, назначенный окну представления библиотекой MFC. В результате из-за переходов между этими двумя формами при перемещении указателя возникает неприятное мерцание.

Вторая: если пользователь выбирает на панели управления темный цвет «Window», линии, нарисованные в окне представления, становятся невидимыми или едва заметными. При создании окна MFC присваивает ему установки, задающие цвет фона с использованием текущего цвета «Window». Однако программа всегда выводит черные линии.

Обе проблемы можно решить, добавив необходимые строки в функцию `PreCreateWindow` класса `CMiniDrawView`. При генерации программы **Мастер приложений** определяет шаблон функции `CMiniDrawView::PreCreateWindow`, переопределяющей виртуальную функцию `PreCreateWindow` класса `CView`, которую MFC вызывает непосредственно перед созданием окна представления.

Чтобы настроить окно представления `MiniDraw`, добавьте операторы, выделенные курсивом, в функцию `PreCreateWindow` в файле `MiniDrawView.cpp`:

```
BOOL CMiniDrawView::PreCreateWindow(CREATESTRUCT& cs)
{
    m_ClassName=AfxRegisterWndClass(
        CS_HREDRAW|CS_VREDRAW, // стили окна
        0, // без указателя;
        (HBRUSH)::GetStockObject (WHITE_BRUSH), // задать чисто белый фон;
        0); // без значка
    cs.lpszClass = m_ClassName;
    return CView::PreCreateWindow(cs);
}
```

В функцию `PreCreateWindow` передается ссылка на структуру `CREATESTRUCT`, поля которой хранят свойства окна, задаваемые библиотекой MFC при его создании (координаты окна, его стили и т. д.). Поле `lpszClass` хранит имя класса окна Windows, но это не класс языка C++, а структура данных, сохраняющая набор общих свойств окна. В добавленном фрагменте вызывается функция `AfxRegisterWndClass`, создающая новый класс окна, а затем присваивается имя класса полю `lpszClass` структуры `CREATESTRUCT`. Таким образом, окно представления создается с настраиваемыми свойствами, сохраняемыми внутри данного класса. Обратите внимание: `AfxRegisterWndClass` – глобальная функция, предоставляемая библиотекой MFC.

При вызове функции `AfxRegisterWndClass` устанавливаются следующие параметры:

- первый параметр задает стили `CS_HREDRAW` и `CS_VREDRAW`, позволяющие перерисовать окно при изменении его размеров (обычно окна представления создаются с использованием этих двух стилей);

- второй параметр задает форму указателя, автоматически отображаемого в окне Windows. Этому параметру присваивается значение «0», поэтому Windows не пытается его отобразить с помощью функции `OnMouseMove`. Таким образом, нежелательное мерцание устраняется;

- третий параметр определяет стандартную белую кисть, используемую для заливки фона окна представления. В результате цвет фона окна всегда будет белым, а черные линии – видимыми, независимо от цвета «Window», выбранного в панели управления;

– четвертый параметр определяет значок окна. Так как окно представления его не отображает, параметру присваивается значение «0» (значок программы задается для главного окна).

**Удаление данных документа.** Каждый раз, когда пользователь выбирает в меню File команду New, MFC (а именно, функция OnFileNew класса CWinApp) вызывает виртуальную функцию CDocument::DeleteContents для удаления содержимого текущего документа перед инициализацией нового.

В последующих версиях программы MiniDraw эта функция будет также вызываться перед открытием существующего документа.

Чтобы удалить данные, сохраняемые этим классом, необходимо написать новую версию этой функции в виде члена класса документа.

Переопределение виртуальной функции является общераспространенным и эффективным способом настройки MFC. Чтобы сгенерировать объявление и оболочку функции DeleteContents, выполните следующие действия.

Выберите закладку **Окно классов**. Найдите класс CMiniDrawDoc и вызовите всплывающее меню, в котором выберите команду **Свойства**, после чего на экране появится окно **Свойства**. В нем выберите закладку **Переопределения**, как это показано на рис. 1.5.

Выбор **Переопределения** переопределяет виртуальную функцию, которую класс CMiniDrawDoc наследует от CDocument. Справа от идентификатора DeleteContents во всплывающем списке выберите команду **<Add> DeleteContents**. Выбор команды объявляет виртуальную функцию в определении класса CMiniDrawDoc в файле MiniDrawDoc.h, вносит ее определение в файл MiniDrawDoc.cpp. После добавления виртуальной функции справа от идентификатора DeleteContents во всплывающем списке выберите команду **<Edit Code>**, после чего откроется файл MiniDrawDoc.cpp и отобразит только что созданную функцию DeleteContents, чтобы можно было добавить ее код.

Добавьте в функцию DeleteContents операторы, выделенные курсивом:

```
void CMiniDrawDoc::DeleteContents()
{
    int Index=(int)m_LineArray.GetSize();
    while(Index--)
        delete m_LineArray.GetAt (Index);
    m_LineArray.RemoveAll ();
    CDocument::DeleteContents();
}
```

В коде, добавленном в определение функции DeleteContents, сначала происходит обращение к функции CObArray класса GetSize, чтобы получить количество указателей класса CLine, сохраненных в данный момент

объектом `m_LineArray`. Затем при вызове функции `CTypedPtrArray::GetAt` выбирается каждый указатель, а оператор `delete` используется для удаления каждого соответствующего объекта класса `CLine` (объекты класса `CLine` создавались с использованием оператора `new`). Наконец, для удаления всех указателей, сохраненных в данное время в массиве `m_LineArray`, вызывается функция `RemoveAll` класса `CObArray`.

После вызова функции `DeleteContents` библиотека MFC (косвенным образом) удаляет окно представления и вызывает функцию `OnDraw`. Однако функция `OnDraw` не отображает линии, потому что они были удалены из класса документа. Окончательный результат – удаление данных документа командой **Создать** (из меню **Файл**) и очистка окна представления перед созданием нового рисунка.

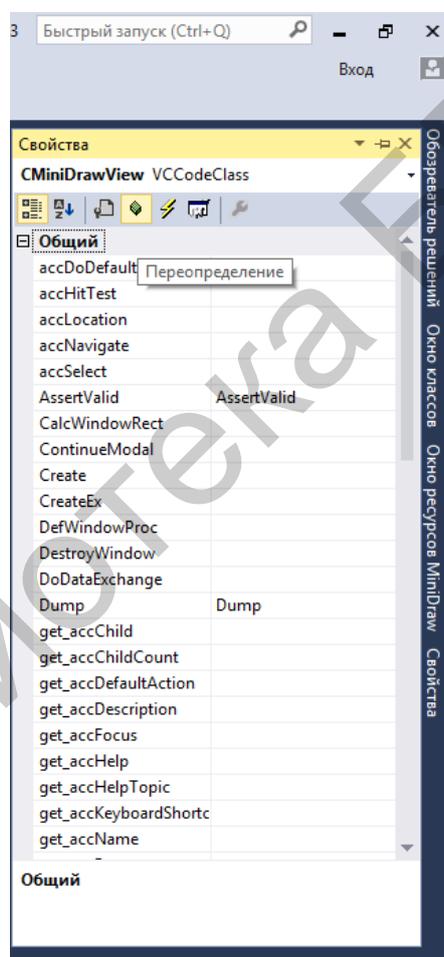


Рис. 1.5. Окно выбора закладки **Переопределение**

**Реализация команд меню.** Теперь воспользуемся **Мастером классов** для реализации двух команд, добавленных в меню **Правка: Удалить всё** и **Назад**.

Выполните следующие действия для определения обработчика сообщения, получающего управление при выборе команды `Delete All`.

Выберите закладку **Окно классов**. Найдите класс CMiniDrawDoc и вызовите всплывающее меню, в котором выберите команду **Свойства**, после чего на экране появится окно **Свойства**. В окне **Свойства** выберите закладку **События**, как представлено на рис. 1.6.

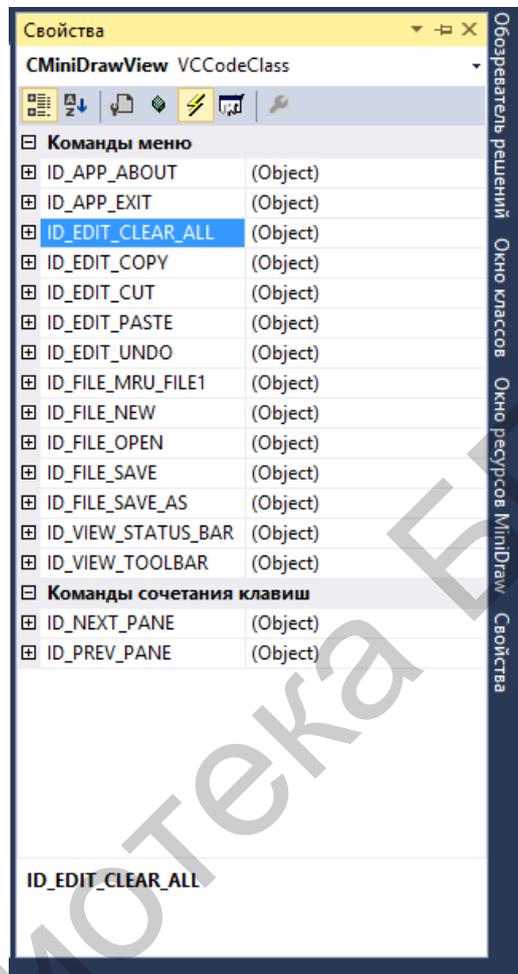


Рис. 1.6. Окно выбора закладки **События**

В списке выберите идентификатор ID\_EDIT\_CLEAR\_ALL. Сразу после выбора идентификатора в списке **События** отобразятся идентификаторы двух типов сообщений (как показано на рис. 1.7), которые эта команда меню 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. Обратите внимание: идентификаторы **COMMAND** и **UPDATE\_COMMAND\_UI** указывают на два типа командных сообщений, которые могут генерироваться объектом пользовательского интерфейса.

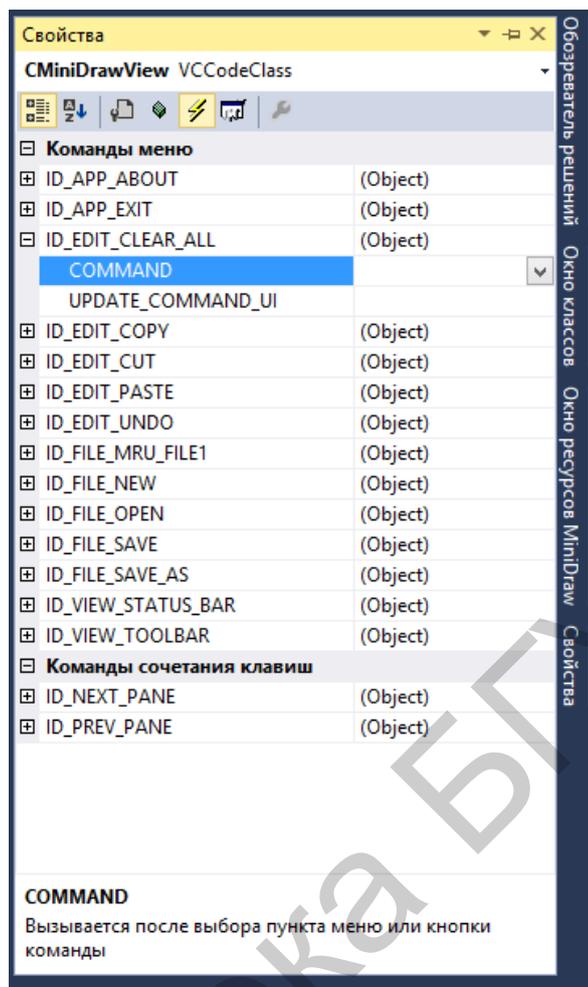


Рис. 1.7. Окно выбора обработчиков для пункта меню *Удалить всё*

Справа от идентификатора COMMAND обработчика во всплывающем списке выберите команду **<Add> OnEditClearAll**, после чего добавляется объявление функции внутри определения класса CMiniDrawDoc в файле MiniDrawDoc.h, минимальное определение функции в файл MiniDrawDoc.cpp, а также генерируется код для добавления функции в схему обработки сообщений класса документа.

Справа от идентификатора обработчика во всплывающем списке выберите команду **<Edit Code>**, после чего откроется файл MiniDrawDoc.cpp и отобразит только что созданную функцию OnEditClearAll, чтобы можно было добавить ее код.

Добавьте в функцию OnEditClearAll операторы, выделенные курсивом:

```
void CMiniDrawDoc::OnEditClearAll()
{
    DeleteContents();
    UpdateAllViews(0);
}
```

Вызов ранее определенной функции DeleteContents приводит к удалению содержимого документа. Далее функция UpdateAllViews класса CDocument удаляет текущее содержимое окна представления.

Следующим этапом будет определение обработчика сообщения UPDATE\_COMMAND\_UI, посылаемого при первом открытии пункта меню, содержащего команду *Удалить всё* (меню *Правка*). Так как это сообщение посылается до того, как меню станет видимым, обработчик может использоваться для инициализации команды в соответствии с текущим состоянием программы. Обработчик делает команду *Удалить всё* доступной, если документ содержит одну или более линий, и недоступной, если документ их не содержит. Чтобы сделать это, следуйте описанным действиям, однако в списке Events выберите идентификатор UPDATE\_COMMAND\_UI, после чего добавится функция с именем OnUpdateEditClearAll. После выбора команды *<Edit Code>* добавьте в эту функцию нижеприведенный код:

```
void CMiniDrawDoc::OnUpdateEditClearAll(CCmdUI *pCmdUI)
{
    pCmdUI->Enable((int)m_LineArray.GetSize());
}
```

Функции OnUpdateEditClearAll передается указатель на объект класса CCmdUI – это MFC-класс, предоставляющий функции для инициализации команд меню и других объектов пользовательского интерфейса. Добавленный код вызывает функцию Enable класса CCmdUI. Она делает доступной команду меню *Удалить всё*, если документ содержит хотя бы одну линию. В противном случае блокирует команду, которая отображается затененной серым цветом, и пользователь не может ее выбрать. Таким образом, функцию OnEditClearAll нельзя вызвать, если документ пуст.

**Инициализация команд меню.** В функцию, которая обрабатывает сообщение UPDATE\_COMMAND\_UI команды меню, передается указатель на объект класса CCmdUI, связанный с выбранной командой. Класс CCmdUI предоставляет четыре функции, которые можно использовать для инициализации команд: Enable, SetCheck, SetRadio и SetText.

Чтобы сделать команду доступной, в функцию Enable передается значение TRUE, а для блокирования команды – FALSE. Например:

```
virtual void Enable(BOOL bOn=TRUE);
```

В функцию SetCheck можно передать значение «1», чтобы сделать пункт меню выбранным, или значение «0», чтобы отменить выбор. Например:

```
virtual void SetCheck (int nCheck=1);
```

Обычно команда меню, представляющая какую-либо функциональную возможность программы, выделяется, если данная возможность активизирована.

Чтобы отметить пункт меню с помощью специального маркера (кружка), можно передать значение TRUE функции SetRadio, а чтобы удалить маркер – значение FALSE. Например:

```
virtual void SetRadio(BOOL bOn=TRUE);
```

Наконец, чтобы изменить надпись команды меню, можно вызвать функцию SetText:

```
virtual void SetText(LPCTSTR lpszText);
```

Здесь lpszText – указатель на новую строку текста. Например, если предыдущее действие состояло в удалении текста, то для замены команды *Назад* можно вызвать функцию SetText.

**Обработка команды Назад.** Последняя задача состоит в реализации функции обработки команды *Назад* меню *Правка*.

Сначала необходимо определить функцию, получающую управление, когда пользователь выбирает команду *Назад*. Чтобы сделать это, необходимо выполнить действия, описанные в пункте **Реализация команд меню**. В списке *События* выберите идентификатор ID\_EDIT\_UNDO. После чего создастся функция с именем OnEditUndo. Добавьте в эту функцию следующий код:

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

Для получения индекса последней линии в добавленном коде сначала вызывается функция GetUpperBound класса CObArray. Затем с целью получения указателя на объект класса CLine для последней линии вызывается функция CTypedPtrArray::GetAt, а для удаления этого объекта используется оператор delete. И, наконец, вызывается функция UpdateAllViews, которая удаляет окно представления и вызывает функцию CMiniDrawView::OnDraw. После этого обработчик OnDraw перерисовывает линии, оставшиеся в документе. Обратите внимание: при многократном

выборе команды **Назад** обработчик OnEditUndo продолжает удалять линии до тех пор, пока не останется ни одной.

Теперь необходимо определить функцию инициализации команды меню **Назад**. Чтобы сделать это, необходимо выполнить действия, описанные в пункте **Реализация команд меню**. В списке **События** выберите идентификатор ID\_EDIT\_UNDO, а в списке **Сообщения** – идентификатор сообщения UPDATE\_COMMAND\_UI. После чего создастся функция с именем OnUpdateEditUndo. Добавьте в эту функцию следующий код:

```
void CMiniDrawDoc::OnUpdateEditUndo(CCmdUI *pCmdUI)
{
pCmdUI->Enable((int)m_LineArray.GetSize());
}
```

Работа этой функции аналогична работе функции OnUpdateEditClearAll, описанной ранее. Она делает команду **Назад** доступной при наличии хотя бы одной линии для удаления.

### Задания

1. Разработать программу, позволяющую рисовать окружности.
2. Разработать программу, позволяющую рисовать прямоугольники.
3. Разработать программу, позволяющую рисовать окружности различными цветами.
4. Разработать программу, позволяющую рисовать прямоугольники различными цветами.
5. Разработать программу, позволяющую рисовать линии различными цветами.
6. Разработать программу, позволяющую рисовать окружности и заливать их различными цветами.
7. Разработать программу, позволяющую рисовать прямоугольники и заливать их различными цветами.

## ЛАБОРАТОРНАЯ РАБОТА №2 СОХРАНЕНИЕ И ВОССТАНОВЛЕНИЕ СОСТОЯНИЯ ОБЪЕКТОВ КЛАССОВ

Цель работы – ознакомиться с процессом сохранения и восстановления внутреннего представления объектов классов.

### Методические указания

В лабораторной работе изучаются принципы сохранения и загрузки данных документа с файлов на диске. Для демонстрации базовых методов ввода-вывода рассмотрена методика добавления кода, реализующего стандартные команды меню **Файл** (**Создать**, **Открыть...**, **Сохранить** и **Сохранить как...**), в программу, написанную в лабораторной работе №1.

В редакторе меню откройте меню **Файл** (рис. 2.1). Ниже для каждой команды в табл. 2.1 приведены идентификатор, надпись и другие свойства.

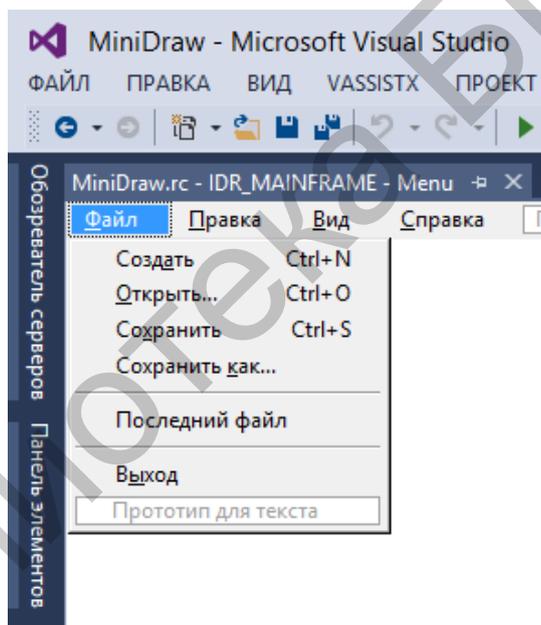


Рис. 2.1. Меню **Файл**

Таблица 2.1

Свойства пунктов меню **Файл**

Идентификатор пункта	Надпись	Другие параметры
ID_FILE_OPEN	&Открыть...\tCtrl-O	Отсутствуют
ID_FILE_SAVE	Со&хранить\tCtrl+S	Отсутствуют
ID_FILE_SAVE_AS	Сохранить &как...	Отсутствуют
Отсутствует	Отсутствует	Разделитель
ID_FILE_MRU_FILE1	Последний файл	Недоступны

Если в программе открыт хотя бы один файл, то MFC заменяет надпись *Последний файл* именем последнего открытого файла. MFC будет добавлять в меню *Файл* имена последних использованных файлов. При создании программы *Мастер приложений* устанавливает максимальное количество последних использованных файлов, равное 4.

Следующий этап – изменение строкового ресурса программы для определения стандартного расширения файлов, отображаемых в диалоговых окнах *Открыть* и *Сохранить как...* Для этого откройте редактор строк, выполнив двойной щелчок на элементе String Table графа *Окно ресурсов*.

Первая строка в окне редактора имеет идентификатор IDR\_MAINFRAME. Она создана *Мастером приложений* и содержит информацию, относящуюся к программе. Ее текущее значение представлено на рис. 2.2.

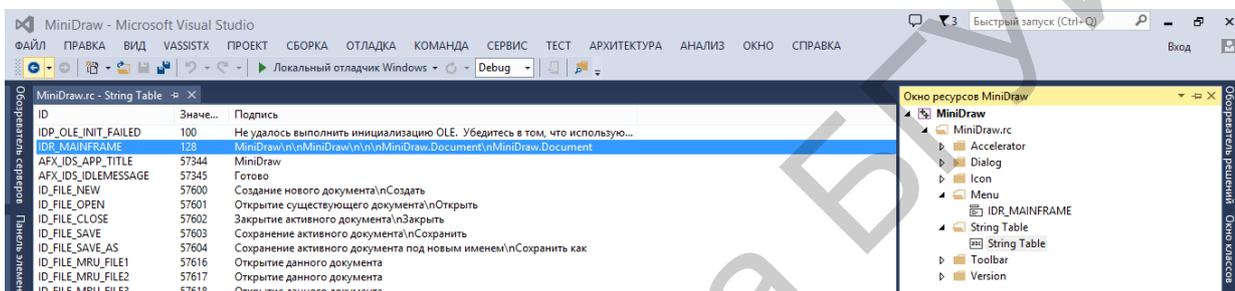


Рис. 2.2. Таблица строк

Чтобы модифицировать строку, откройте диалоговое окно *Свойства* выделенной строки. Измените содержимое поля *Надпись* следующим образом:

```
MiniDraw\n\nMiniDraw\nMiniDrawFiles (*.drw)\n.n.drw\nMiniDraw.Docum
ent\nMiniDraw.Document
```

При редактировании строки не нажимайте клавишу Enter. Когда текст достигнет края текстового поля, тогда он автоматически будет перенесен на следующую строку.

Вставьте строку MiniDraw Files (\*.drw), отображаемую в списке *Тип* файла диалогового окна *Открыть* (или *Сохранить как...*) и определяющую стандартное расширение файлов программы. Затем вставьте (.drw) – стандартное расширение файлов. Если в процессе выполнения программы MiniDraw расширение файла при открытии или сохранении не указано, то в диалоговых окнах *Открыть* и *Сохранить как...* отобразится список всех файлов со стандартными расширениями, а в диалоговом окне *Сохранить как...* стандартное расширение файла будет добавлено к его имени.

**Задание расширения файлов в программе.** Приведенные инструкции относятся только к заданию стандартного расширения файлов в

существующей программе. Это можно также сделать при создании приложения *Мастером приложений*. В диалоговом окне *Мастера приложений* щелкните на *Свойства шаблона*. Введите в поле *Расширение файла* стандартное расширение файла (без точки), например drw. После ввода стандартного расширения *Мастер приложений* автоматически введет описание расширения в поле *Имя фильтра* (например, MiniDraw Files (\*.drw)) (рис. 2.3).

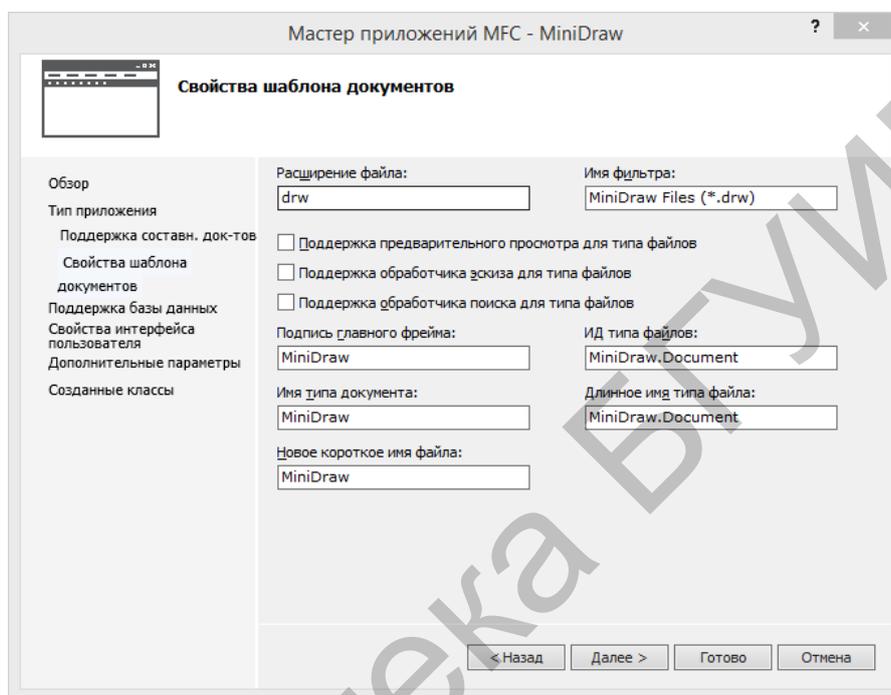


Рис. 2.3. Свойства шаблонов документов

Эта строка отображается в диалоговом окне *Открыть* (или *Сохранить как...*) в списке *Тип файла*. При желании эту строку можно отредактировать. Введите оставшуюся информацию в диалоговые окна *Мастера приложений*.

**Поддержка команд меню Файл.** В лабораторной работе №1 при добавлении команд *Назад* и *Удалить всё* в меню *Правка* для определения их обработчиков использовался *Мастер классов*. Для команд *Создать*, *Открыть...*, *Сохранить* и *Сохранить как...* определять обработчики не требуется, т. к. они предоставляются MFC. В этом случае необходимо написать код для их поддержки. Библиотека MFC также предоставляет обработчики команд для работы с последними использованными файлами в меню *Файл*. Функция OnFileNew класса CWinApp (от которого порождился класс приложения MiniDraw) обрабатывает команду *Создать*. Эта функция вызывает виртуальную функцию DeleteContents для удаления текущего содержимого документа, а затем инициализирует новый документ.

Команда *Открыть...* обрабатывается функцией OnFileOpen класса CWinApp, которая отображает стандартное диалоговое окно *Открыть*. Если

выбрать файл и щелкнуть на кнопке **Открыть**, то OnFileOpen откроет файл для чтения, а затем вызовет функцию Serialize класса документа (CMiniDrawDoc::Serialize). Функция Serialize предназначена для фактического выполнения операции чтения. Функция OnFileOpen сохраняет полный путь к загруженному файлу и отображает имя файла в строке заголовка главного окна.

Функция OnFileSave класса CDocument обрабатывает команду **Сохранить**, а функция OnFileSaveAs класса CDocument – команду **Сохранить как...**. Если документ сохраняется впервые, то функции OnFileSaveAs и OnFileSave начинают работу с отображения стандартного диалогового окна **Сохранить как...**, позволяющего задать имя файла. Эти функции открывают файл для записи, а затем вызывают функцию CMiniDrawDoc::Serialize для выполнения собственно операции записи.

**Сериализация данных документа.** При создании программы MiniDraw *Мастер приложений* определяет в файле MiniDrawDoc.cpp следующую минимальную реализацию функции Serialize:

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

В эту функцию необходимо добавить собственный код для чтения или записи данных. MFC передает функции Serialize ссылку на экземпляр класса CArchive. Если файл открыт для записи (т. е. выбрана команда **Сохранить** или **Сохранить как...**), то функция IsStoring класса CArchive возвращает значение TRUE, а если файл открыт для чтения (т. е. выбрана команда **Открыть...** или команда вызова последнего использованного файла из меню **Файл**), то функция IsStoring возвращает значение FALSE. Следовательно, код операции вывода помещается внутри блока if, а код операции ввода – внутри блока else.

В программе MiniDraw класс документа хранит единственную переменную m\_LineArray, управляющую множеством объектов класса CLine. Переменная m\_LineArray имеет собственную функцию-член Serialize (наследуемую от класса CObArray), которая вызывается для чтения или записи всех объектов класса CLine, хранимых в данной переменной. В результате функцию CMiniDrawDoc::Serialize можно дописать, просто добавив два обращения к функции CObArray::Serialize:

```
void CMiniDrawDoc::Serialize(CArchive& ar)
```

```

{
    if(ar.IsStoring())
    {
        m_LineArray.Serialize(ar);
    }else
    {
        m_LineArray.Serialize(ar);
    }
}

```

При записи данных в файл функция CObArray::Serialize выполняет два основных действия для каждого объекта класса CLine: записывает в файл информацию о классе объекта; вызывает функцию Serialize объекта, записывающую данные объекта в файл. При чтении данных из файла функция CObArray::Serialize выполняет два действия для каждого объекта класса CLine: читает информацию класса из файла, динамически создает объект соответствующего класса (CLine) и сохраняет указатель на объект; вызывает функцию Serialize объекта для чтения данных из файла во вновь созданный объект.

Необходимо обеспечить поддержку сериализации класса CLine. Для этого следует включить в его определение два макроса (DECLARE\_SERIAL, а также IMPLEMENT\_SERIAL) и определить конструктор класса по умолчанию. Эти макрокоманды и конструктор позволяют функции CObArray::Serialize сохранить информацию класса в файле, а затем использовать ее для динамического создания объекта класса. Макрокоманды и конструктор обеспечивают выполнение первого пункта двух приведенных выше алгоритмов с помощью функции CObArray::Serialize().

Добавьте макрокоманду DECLARE\_SERIAL и конструктор по умолчанию в определение класса CLine в файле MiniDrawDoc.h, как показано ниже.

```

class CLine : public CObject
{
protected:
    int m_X1, m_Y1, m_X2, m_Y2;
    CLine() {}
    DECLARE_SERIAL(CLine)
public:
    //оставшаяся часть определения класса CLine
}

```

Имя класса передается макросу DECLARE\_SERIAL как параметр. В файле MiniDrawDoc.cpp макрокоманда IMPLEMENT\_SERIAL добавляется сразу перед определением функции CLine::Draw:

```

IMPLEMENT_SERIAL(CLine, CObject, 1)
void CLine::Draw(CDC *pDC)

```

```

{
    pDC->MoveTo (m_X1, m_Y1) ;
    pDC->LineTo (m_X2, m_Y2);
}

```

Первый параметр, переданный макросу IMPLEMENT\_SERIAL, – это имя самого класса, а второй – имя базового класса. Третий параметр – это номер версии, идентифицирующий отдельную версию программы. Он сохраняется внутри записанного файла, прочитать который может только программа, указавшая такой же номер. Номер версии предотвращает чтение данных программой другой версии. Для текущей версии MiniDraw задан номер 1. В более поздних версиях он увеличивается при каждом изменении формата данных. Нельзя задавать номер версии – 1.

Второе действие, необходимое для сериализации класса CLine, – это добавление в класс CLine функции Serialize, вызываемой в методе CObArray::Serialize для чтения или записи данных каждой строки. Добавьте объявление функции Serialize в раздел public определения класса CLine в файле MiniDrawDoc.h:

```

public:
    CLine (int X1, int Y1, int X2, int Y2) {
        m_X1=X1; m_Y1=Y1; m_X2=X2; m_Y2=Y2;
    }
    void Draw (CDC *pDC);
    virtual void Serialize(CArchive& ar);

```

Добавьте определение функции Serialize в файл реализации с именем MiniDrawDoc.cpp после определения CLine::Draw:

```

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

```

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

Операции чтения и записи выполняет функция CLine::Serialize. Функция Serialize использует перегруженные операторы << и >> для записи переменных класса CLine в файл и для чтения их из файла соответственно. Для переменных, являющихся объектами класса, вызывается функция-член Serialize их собственного класса.

**Установка флага изменений.** Класс CDocument поддерживает флаг изменений, показывающий, содержит ли документ несохраненные данные.

MFC проверяет этот флаг перед вызовом функции DeleteContents класса документа для удаления данных. MFC вызывает функцию DeleteContents перед созданием нового документа открытием уже существующего или выходом из программы. Если флаг содержит значение TRUE (в документе имеются несохраненные данные), то выводится соответствующее сообщение.

Класс CDocument устанавливает значение флага в FALSE, когда документ открыт и сохранен. Для установки флага в TRUE (при каждом изменении документа) вызывается функция CDocument::SetModifiedFlag. Добавьте функции SetModifiedFlag в функцию AddLine в файле MiniDrawDoc.cpp:

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

Теперь сделайте то же самое для функции OnEditClearAll в том же файле:

```
void CMiniDrawDoc::OnEditClearAll()
{
    DeleteContents();
    UpdateAllViews(0);
    SetModifiedFlag();
}
```

И, наконец, добавьте вызов функции SetModifiedFlag в обработчик OnEditUndo в файле MiniDrawDoc.cpp:

```
void CMiniDrawDoc::OnEditUndo()
{
    int Index=m_LineArray.GetUpperBound();
    if(Index>-1){
        delete m_LineArray.GetAt (Index);
        m_LineArray.RemoveAt (Index);
    }
    UpdateAllViews(0);
    SetModifiedFlag();
}
```

**Поддержка технологии «drag-and-drop».** Если программа поддерживает традиционную технологию «drag-and-drop», можно открывать файл, перемещая объект файла из папки Windows (а также из окна программы Explorer или любого другого окна, поддерживающего эту операцию) и отпуская его в окне программы.

Для поддержки операции перетаскивания в программе MiniDraw вызовите функцию `CWnd::DragAcceptFiles` для объекта главного окна. Поместите это обращение внутри функции `InitInstance` класса приложения в файле `MiniDraw.cpp` после вызова `UpdateWindow`:

```
BOOL CMiniDrawApp::InitInstance()
{
    // другие операторы
    if(!ProcessShellCommand(cmdInfo))
        return FALSE;
    m_pMainWnd->ShowWindow(SW_SHOW);
    m_pMainWnd->UpdateWindow();
    m_pMainWnd->DragAcceptFiles();
    return TRUE;
}
```

Объект приложения содержит переменную `m_pMainWnd` (определенную в классе `CWinThread`, базовом для `CWinApp`), являющуюся указателем на объект главного окна. Функция `InitInstance` использует этот указатель для вызова функции `DragAcceptFiles`. Обращение к ней помещается после вызова функции `ProcessShellCommand`, т. к. внутри последней создается главное окно и присваивается значение переменной `m_pMainWnd`.

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

**Регистрация типа файла.** В системный реестр Windows следует добавить информацию, позволяющую открыть файл программы MiniDraw (т. е. файл с расширением `.drw`), выполняя двойной щелчок на файле в папке Windows. Для этого вызовите функции `EnableShellOpen` и `RegisterShellFileTypes` класса `CWinApp` из определения функции `InitInstance` в файле `MiniDraw.cpp`:

```
BOOL CMiniDrawApp::InitInstance()
{
    // другие операторы...
    AddDocTemplate(pDocTemplate);
    EnableShellOpen();
    RegisterShellFileTypes();
    // Анализ командной строки с целью поиска //
    команд оболочки, DDE, открытия файлов
    CCommandLineInfo cmdInfo; //
    другие операторы...
}
```

Эти функции создают в реестре Windows связь между стандартным расширением файла программы MiniDraw (`.drw`) и самой программой. Объект, представляющий любой файл с этим расширением, отображает значок программы MiniDraw, а двойной щелчок на объекте запускает

программу MiniDraw, если она еще не запущена, и открывает файл в этой программе. Такая связь остается в реестре до тех пор, пока она не будет изменена явным образом.

Обращения к функциям EnableShellOpen и RegisterShellFileTypes помещаются после вызова функции AddDocTemplate, добавляющей шаблон документа в приложение, чтобы информация о стандартном расширении файла и типе документа была доступна объекту приложения (стандартное расширение вводится в строковый ресурс с идентификатором IDR\_MAINFRAME, который передается в шаблон.)

### **Задание**

Дополнить программу, разработанную в лабораторной работе №1, возможностями сохранения и восстановления данных.

Библиотека БГУМР

## ЛАБОРАТОРНАЯ РАБОТА №3 СТАНДАРТНЫЕ ЭЛЕМЕНТЫ УПРАВЛЕНИЯ

Цель работы – научиться использовать в приложении диалоговые окна и стандартные элементы управления.

### Методические указания

На первом этапе создадим диалоговое окно и назовем приложение SDI. Процесс создания ресурсов для диалоговых окон подробно рассмотрен в прил. 2.

Установите элементы управления на поверхности диалога. Для этого щелкните мышью на пиктограмме соответствующего элемента управления, а затем – на форме в том месте, где будет расположен этот элемент. Если посчитаете нужным, «ухватитесь» указателем мыши за один из размерных маркеров вокруг изображения элемента. С помощью его передвижения можно изменять границы размеров элемента (рис. 3.1, 3.2).

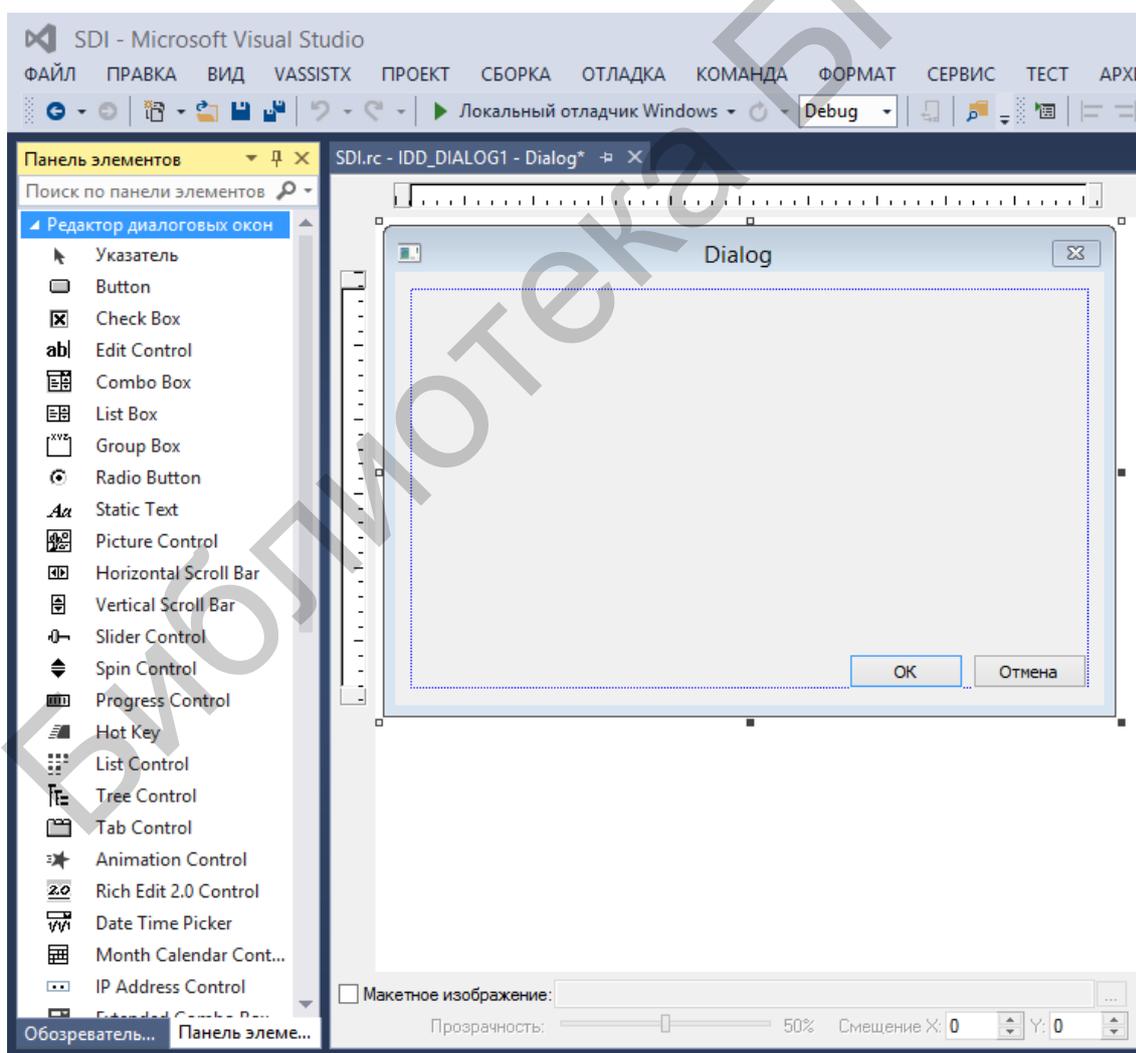


Рис. 3.1. Панель элементов

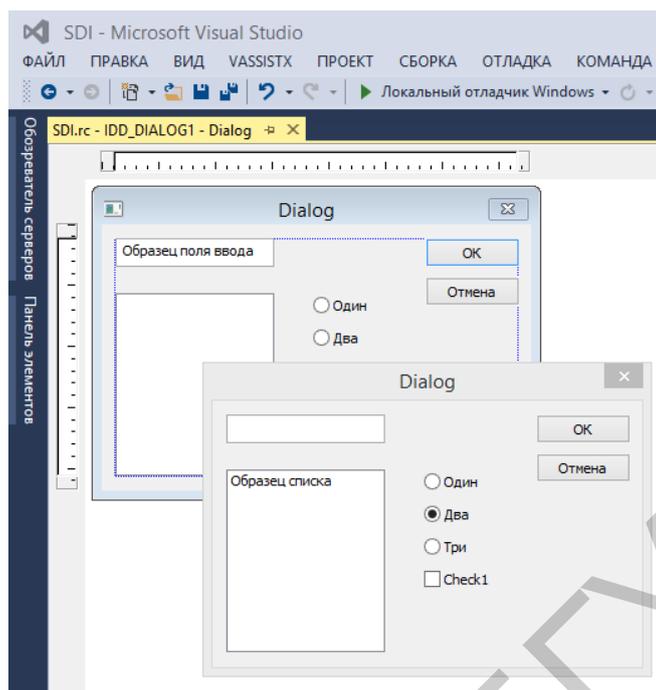


Рис. 3.2. Диалоговое окно с элементами управления

Добавьте элемент типа флажок и три переключателя в диалоговое окно, чтобы оно приняло вид, изображенный на рис. 3.2. В поле **Надпись** измените надписи для переключателей. Пусть это будут: **Один**, **Два** и **Три**. Чтобы выровнять включенные в окно элементы (выстроить их в колонку), выберите один из них, а затем, нажав и удерживая клавишу Ctrl, выберите по очереди остальные. После этого выберите в меню **Lay-Out=>Align Control Left**, затем выберите команду **Lay-Out=>Space Evenly=>Down** (**Размещение=>Подравнять интервал=>Вниз**). Эта команда позволяет установить одинаковый интервал между элементами по вертикали.

Выберите переключатель **Один**, вызовите окно **Свойства** и установите в нем флажок **Group** (**Группа**). Такая установка означает, что переключатель **Один** является первым элементом группы переключателей.

Теперь добавьте в формируемое окно еще и элемент типа **Список**. Установите его слева от переключателей и измените размеры в соответствии с заданием. В то время когда этот элемент еще остается выбранным, вызовите его свойства. В разделе **Поведение** проверьте, не установлен ли пункт **Сортировка** в TRUE. Если этот флажок установлен, то при выполнении программы элементы в списке будут отсортированы по алфавиту.

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 после ее нажатия;
- 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 выбирать элементы из ранее подготовленного набора, но и самостоятельно пополнять его, непосредственно внося необходимый текст в текстовое поле.

#### **Задание идентификаторов диалогового окна и элементов управления.**

Поскольку каждое диалоговое окно в приложении является уникальным объектом, разработчику необходимо 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 окна *Свойства*. Для этого выберите элемент управления или диалог, затем щелкните правой кнопкой мыши и выберите пункт *Свойства*. Затем измените идентификатор ресурса в поле *ИД*.

**Создание ассоциированных переменных.** Для начала необходимо создать класс диалогового окна, для этого щелкните правой кнопкой мыши на форме и выберите пункт контекстного меню *Добавить класс...* В открывшемся диалоговом окне *Мастера добавления классов* в поле *Имя класса* укажите имя нашего класса CMyDialogEx (рис. 3.3). Ассоциированная переменная класса диалогового окна задается следующим образом: выберите элемент управления (добавленный в форму), щелкните правой кнопкой мыши и выберите пункт *Добавить переменную...* (рис. 3.4). Пример, представленный на рис. 3.3, демонстрирует один из вариантов ассоциативной связи. Элементу IDC\_CHECK1 следует присвоить идентификатор переменной m\_check. Нужно проверить, чтобы в раскрывающемся списке *Категория* было выбрано Value.

Если вы раскроете список *Тип переменной*, то увидите, что вам предоставлен единственный «свободный» 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 ОК для завершения процедуры.

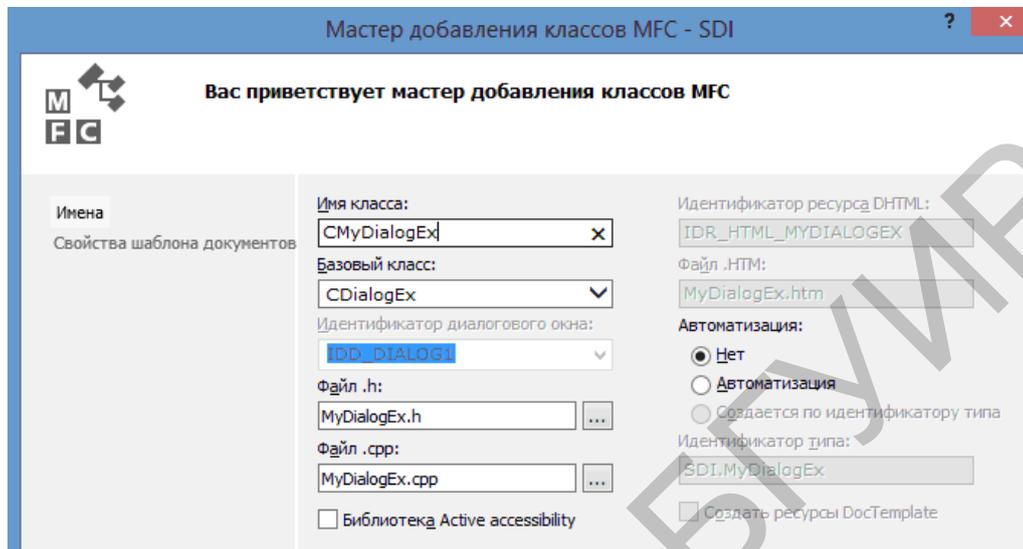


Рис. 3.3. Добавление класса формы

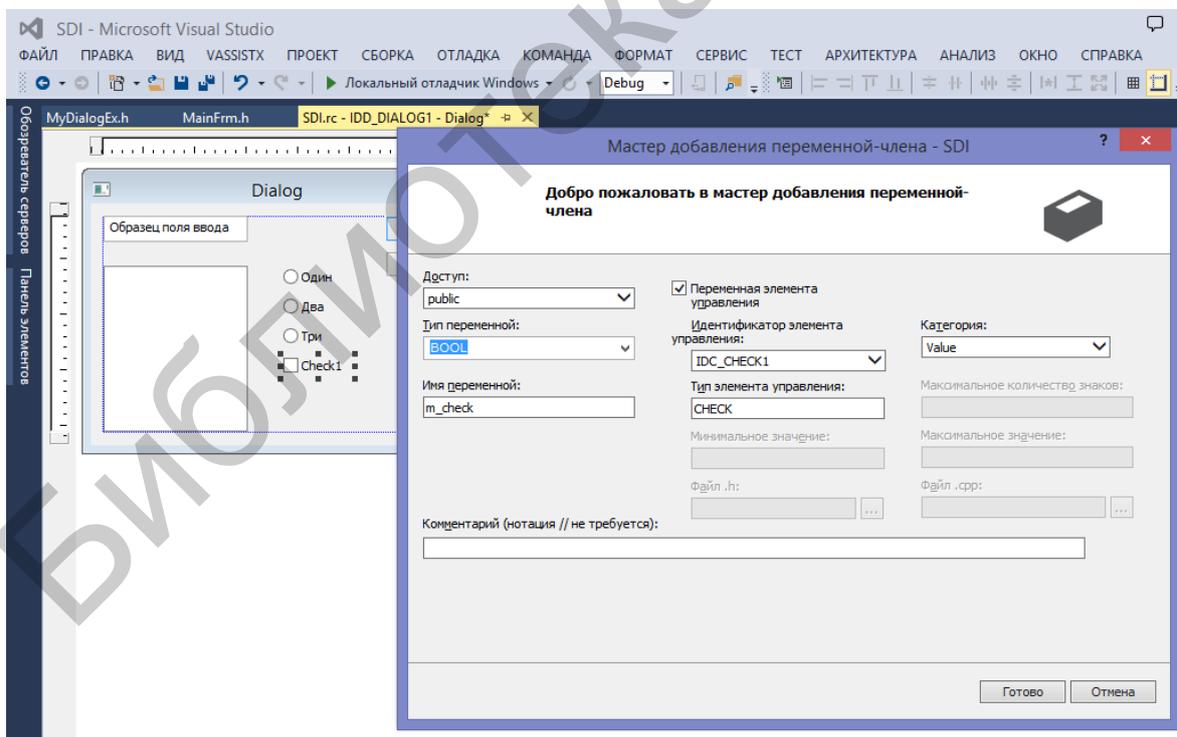


Рис. 3.4. Диалог установки идентификатора члена-переменной класса, ассоциированного с некоторым элементом управления

Ниже перечислены типы переменных, которые могут быть ассоциированы с тем или иным типом элемента управления:

- текстовые поля – как правило, строковый тип (CString), но иногда и другие – int, float и long;
- кнопки – int;
- флажки – BOOL;
- переключатели – int;
- список – строковый тип;
- поле со списком – строковый тип;
- полоса прокрутки – int.

После добавления переменной можно установить параметры, которые могут быть использованы для проверки достоверности ввода данных. Например, если речь идет о переменной, связанной с текстовым полем, можно в поле *Maximum Characters* (*Максимум символов*) установить максимальную длину вводимой строки.

Свяжите таким же образом значение, которое содержится в элементе IDC\_EDIT1, с членом-переменной m\_edit типа CString. Для члена-переменной m\_edit установите максимальное значение этого параметра равным 10. 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.

В классе, связанном с диалогом, присутствует функция OnOK(), которая унаследована от базового класса CDialog, классом-наследником которого является наш диалог. Помимо прочего в нем находится функция DoDataExchange(). Вот как она выглядит в настоящий момент:

```
void CSDIDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    DDX_Check(pDX, IDC_CHECK1, m_check);
    DDX_Control(pDX, IDC_LIST1, m_list);
    DDX_Radio(pDX, IDC_RADIO1, m_radio);
    DDX_Text(pDX, IDC_EDIT2, m_edit);
    DDV_MaxChars(pDX, m_edit, 10);
}
```

Все функции, имена которых начинаются с DDX, выполняют обмен данными. Вторым аргументом каждой функции является идентификатор элемента управления, а третьим – переменная класса. Именно таким образом *Мастер классов* устанавливает соответствие между элементами управления и членами класса диалогового окна.

Организация вывода диалогового окна на экран. Выберите закладку Solution в рабочей зоне проекта, раскройте пункт проекта SDI, а в нем – SDI.cpp. Перейдите в самое начало файла и после уже имеющихся директив #include вставьте еще одну:

```
#include "MyDialogEx.h"
```

Теперь при трансляции компилятор будет знать, где взять информацию о классе CMyDialogEx.

Найдите функцию-член InitInstance(). Эта функция вызывается при любом запуске приложения. Перейдите в конец текста функции CSDIApp::InitInstance() в файле SDI.cpp и добавьте перед окончанием текста функции (перед оператором return) следующие строки:

```
CMyDialogEx 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 программы – экземпляр класса диалогового окна. Он устанавливает параметры по умолчанию для двух элементов управления – флажка и текстового поля. Сам по себе вывод диалогового окна производится функцией DoModal(), которая возвращает числовое значение – IDOK, если пользователь вышел из окна, нажав ОК, и IDCANCEL, если выход произошел после нажатия Cancel. Затем в приведенном фрагменте формируется сообщение, которое выводится на экран функцией AfxMessageBox().

**Использование элемента управления типа список.** Работать со списками сложнее, поскольку реальным объектом список становится тогда, когда диалоговое окно уже выведено на экран. Нельзя вызывать функции-члены списка до тех пор, пока не будет порожден экземпляр класса диалогового окна. Таким образом, инициализировать список – заполнить его элементами-строками – и анализировать, какая строка выбрана, можно только на том участке программы, который выполняется во время присутствия диалогового окна на экране.

Когда наступает время инициализировать диалоговое окно, перед выводом его на экран вызывается функция-член класса CDialog OnInitDialog(). Для того чтобы включить эту функцию в члены класса, необходимо воспользоваться услугами *Мастера классов*.

В окне *Мастера классов* щелкните правой кнопкой мыши на CMyDialogEx и в контекстном меню выберите *Мастер классов*. На экране появится окно *Мастер классов*. Щелкните по закладке *Виртуальные функции* (рис. 3.5). Далее из списка *Виртуальные функции* необходимо выбрать OnInitDialog и щелкнуть на кнопке *Добавить функцию*. Щелкните на OnInitDialog и затем щелкните на кнопке *Изменить код*, с тем чтобы увидеть текст программы.

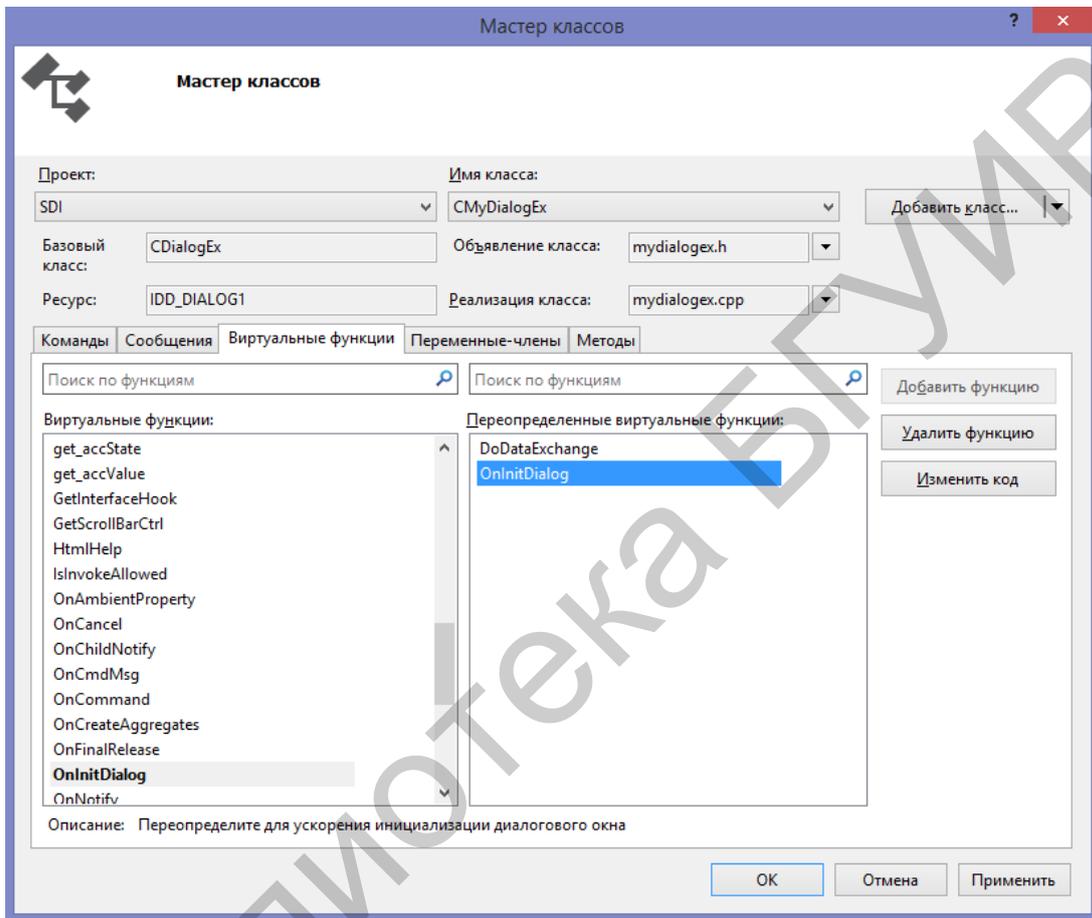


Рис. 3.5. Добавление OnInitDialog

Вставьте вызовы функций-членов списка.

```

BOOL CSDIDialog:: OnInitDialog()
{CDialog::OnInitDialog();
    m_list.AddString(L"First String");
    m_list.AddString(L"Second String");
    m_list.AddString(L"Third String");
    m_list.AddString(L"Fourth String");
    m_list.SetCurSel(2);
return TRUE; //Возвращает TRUE, если только вы не
// установили фокус ввода на элемент управления
}

```

Для того чтобы организовать вывод в окне сообщения информации о выбранном из списка элементе, нужно также включить специальную член-переменную в класс диалогового окна. В эту переменную при закрытии окна будет записываться значение, к которому затем можно обратиться, несмотря на то, что окно уже закрыто. В **Окне классов** щелкните правой кнопкой мыши на CMyDialogEx и выберите **Добавить=>Добавить переменную....** Заполните реквизиты в окне **Мастера добавления переменных**: имя – m\_selected; тип – CString и щелкните **Готово** (рис. 3.6).

Описание переменной появится в файле заголовка класса диалогового окна. Если планируется, что список должен поддерживать многозначный выбор, то тип введенной переменной будет CStringArray, т. е. эта переменная будет содержать массив выбранных переменных. Член-переменную следует объявлять закрытой (private), а функции доступа к ней открытыми (public). Эта новая переменная будет хранить выбранный пользователем элемент списка.

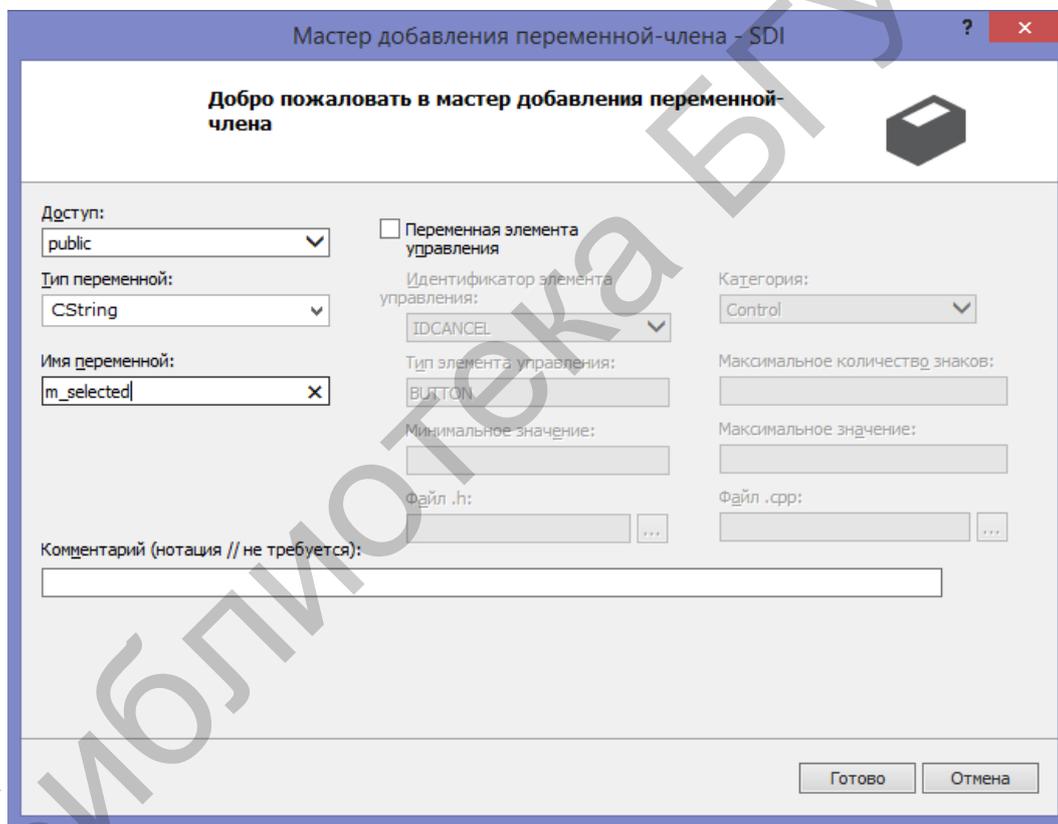


Рис. 3.6. Добавление переменной m\_selected

**Создание обработчиков событий нажатия кнопок ОК и CANCEL.** Простейшим событием является нажатие кнопки в диалоге. Для его создания можно просто дважды щелкнуть левой кнопкой мыши в окне ресурса по соответствующей кнопке. После этого на экране появится код MyDialogEx.cpp и созданная функция:

```
void CSDIDialog::OnBnClickedOk()
```

```

{
    CDialog::OnOK();
}

```

Также можно создать эту функцию вызовом *Мастера классов*, где во вкладке *Команды* для класса CMyDialogEx выбрать из списка IDOK, а затем BN\_CLICKED (обработчик нажатия кнопки) для этого идентификатора, нажав кнопку *Добавить обработчик...* (рис. 3.7). После этого на экране появится диалоговое окно (рис. 3.8).

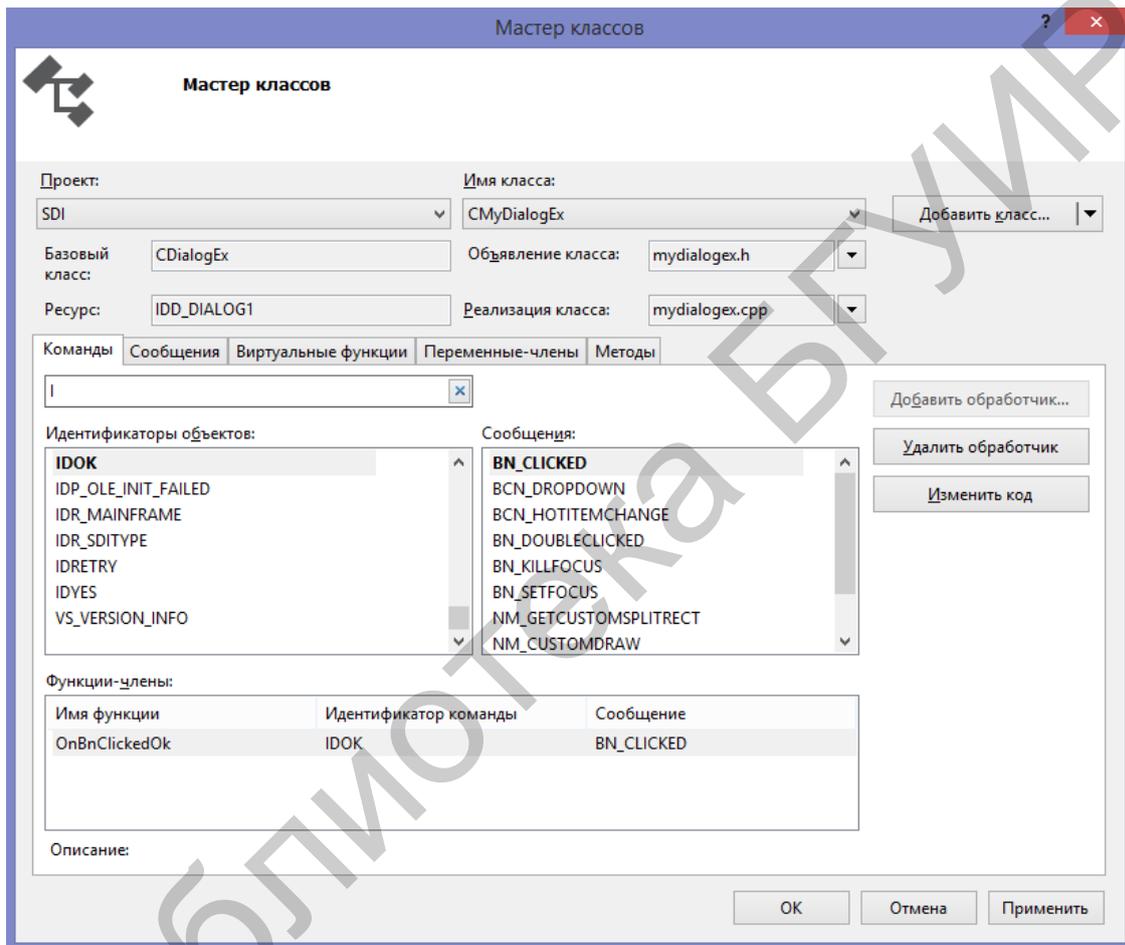


Рис. 3.7. Создание обработчика события нажатия кнопки ОК

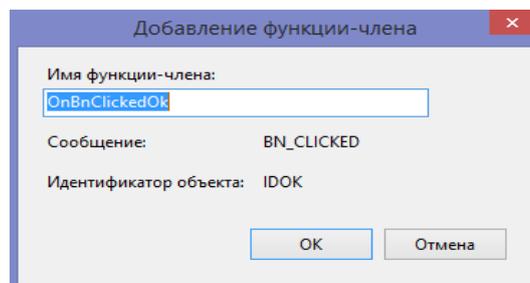


Рис. 3.8. Диалог определения имени функции

По умолчанию *Мастер классов* предлагает очень удачное имя для этой функции обработки события. Далее нажмите кнопку *Изменить код*. В функцию `CMyDialogEx::OnBnClickedOK()` добавьте выделенные курсивом строки:

```
void CSDIDialog:: OnBnClickedOK()
{
    int index = m_list.GetCurSel();
    if (index != LB_ERR)
    {
        m_list.GetText(index, m_selected);
    } else {
        m_selected = "";
    }
    CDialog::On OK();
}
```

Работа этого фрагмента программы начинается с вызова функции-члена класса `GetCurSel()`, которая возвращает константу `LB_ERR` в случае, если не выбран ни один элемент списка или если выбрано более одного элемента. Иначе возвращается индекс выбранного элемента. Функция `GetText()` (член того же класса) переписывает строку выбранного элемента в переменную `m_selected` класса `CMyDialogEx`. Первым аргументом функции является индекс выбранного элемента. После этого вызывается функция `OnBnClickedOK()` базового класса `CDialog`, которая выполняет все стандартные действия по закрытию окна.

Сейчас можно скорректировать текст функции `CSDIApp::InitInstance()` для того, чтобы в выведенном после закрытия окна сообщении было упомянуто, какой выбор сделал пользователь в списке. Эти строки программы будут выполняться независимо от того, каким образом было закрыто окно – нажал пользователь `OK` или `Cancel`. Сначала нужно создать дополнительно функцию, которая обрабатывала бы щелчок на `Cancel`. Такая функция – `OnCancel()` – создается точно таким же образом, как и `OnOK()`, но в правом списке необходимо выделить `IDCANCEL` и согласиться с именем функции `OnBnClickedCancel()`. Как видно из листинга сформированной функции, переменная `m_selected` очищается, поскольку пользователь отказался от диалога с программой.

```
void CMyDialogEx::OnCancel()
{
    m_selected = "";
    CDialog::OnCancel();
}
```

В функцию `CSDIApp::InitInstance()` добавьте следующие строки перед обращением к функции `AfxMessageBox()`:

```
msg += ". List Selection: ";  
msg += dlg.m_selected;
```

Для выбора одного из переключателей в группе диалогового окна в функции `CMyDialogEx::OnInitDialog()` перед `return TRUE`; следует добавить следующий текст:

```
m_radio = 1;  
UpdateData(FALSE);
```

Членом-переменной является `m_radio`, с которым связана группа переключателей. Она (переменная) представляет собой индекс выбранного переключателя в этой группе элементов управления (как всегда, индекс начинается со значения «0»). Значение индекса «1» соответствует второму переключателю в группе. Вызов функции `UpdateData()` в этом фрагменте обновляет содержимое элементов управления диалогового окна соответственно состоянию связанных с ними переменных-членов. Аргумент функции `UpdateData()` указывает направление передачи данных: `UpdateData(TRUE)` обновило бы содержимое переменных соответственно элементам управления, т. е. переписало бы в `m_radio` индекс выбранного в группе переключателя.

В отличие от списка группа переключателей доступна и после того, как диалоговое окно убрано с экрана. Рассмотрим проблему – как преобразовать целое значение индекса в строковое выражение, которое нужно будет добавить в «хвост» текста сообщения в переменную `msg`. Существует множество ее решений, включая функцию-член `Format()` класса `CString`, но в данном случае можно поступить проще – использовать оператор `switch`, поскольку индекс может принимать лишь ограниченное множество значений. В конец текста функции `CSDIApp::InitInstance()` перед вызовом `AfxMessageBox()` добавьте несколько строк:

```
msg += "\r\n";  
msg += "Radio Selection: ";  
switch (dlg.m_radio)  
{ case 0: msg += "0"; break;  
  case 1: msg += "1"; break;  
  case 2: msg += "2"; break;  
  default: msg += "none"; break;  
}
```

Первая из новых строк добавляет в сообщение два специальных символа – перевод каретки `\r` и перевод строки `\n`, которые в совокупности представляют маркер конца строки Windows. В результате дальнейшая часть сообщения `msg` начнется с новой строки.

Запустите процесс компиляции и компоновки проекта, выбрав команду **Сборка=>Собрать решение**. Запустите приложение на выполнение.

После загрузки созданного вами приложения поменяйте что-либо в текстовом поле (например, введите текст hello), а затем щелкните ОК. После этого на экране должно появиться окно сообщения.

Снова запустите приложение, отредактируйте текст в поле и выйдите из окна, щелкнув Cancel. Обратите внимание на сообщение, гласящее, что в поле остался исходный текст hello. Это получилось потому, что MFC не дублирует содержимое текстового поля (как элемента управления) в переменную-член в случае, если пользователь щелкает на Cancel для выхода из окна. И снова, как и в предыдущем эксперименте, закройте приложение.

После того как вы щелкнете ОК, приложение выведет в окне сообщения копию текста, введенного в текстовом поле. Надпись в окне гласит: «*You clicked OK. Edit Box is: hello*». Если вы щелкнете Cancel, приложение проигнорирует любые изменения элементов управления. Надпись в окне гласит: «*You clicked Cancel. Edit Box is: hello*». Также в окнах выводится информация об элементах управления.

## Задания

1. Разработать приложение управления двумя списками, расположенными на диалоге горизонтально. Приложение должно обеспечивать перемещение любого выбранного элемента или содержимого всего списка в другой список, как из левого в правый, так и из правого в левый. Элемент при перемещении должен исчезать из одного списка и появляться в другом. Помимо того приложение должно обеспечивать управление левым списком – добавление нового элемента, редактирование, удаление.

2. Разработать приложение, обеспечивающее возможность множественного выбора элементов из списка. Выбранные элементы должны образовывать строку текста и помещаться в строку редактирования. Предусмотреть возможность вывода сообщения в случае, если суммарное количество символов будет превышать 100.

3. Разработать приложение, реализующее калькулятор. Приложение должно иметь одну строку редактирования, набор кнопок от 0 до 9, кнопки арифметических действий – суммирование, вычитание, деление, умножение.

4. Разработать приложение, реализующее калькулятор. Приложение должно иметь две строки редактирования. Набор переключателей радиокнопок определяет, какое арифметическое действие необходимо выполнить: суммирование, вычитание, деление, умножение.

5. Разработать приложение, обеспечивающее поиск в списке (list box) фрагмента текста. Строки, в которых будет найден искомый фрагмент, должны быть выделены (предполагается, что несколько строк может иметь искомый

фрагмент). Помимо этого, приложение должно обеспечивать управление содержимым списка – добавление нового элемента, редактирование, удаление.

6. Разработать приложение генерации счетчика 100 случайных чисел и вывод их в отсортированном по убыванию порядке. Обеспечить возможность расчета суммы трех наибольших чисел и трех наименьших. Суммируемые числа выделить.

7. Разработать приложение управления двумя списками, расположенными на диалоге горизонтально. Приложение должно обеспечивать перемещение любого количества выбранных элементов. Элемент при перемещении не исчезает, а выделяется. Помимо этого, приложение должно обеспечивать заполнение левого списка 10 строками при запуске приложения.

8. Разработать приложение управления списком. На диалоге установлено два флажка (check box). При первом включенном флажке осуществляется выбор всех нечетных строк, при втором включенном флажке – выбор всех четных строк.

## ЛАБОРАТОРНАЯ РАБОТА №4

### ДОСТУП К БАЗАМ ДАННЫХ С ПОМОЩЬЮ ТЕХНОЛОГИИ ODBC

Цель работы – разработать БД-приложение с использованием ODBC-классов, способное отображать записи, хранящиеся в базе данных, а также обновлять, добавлять, удалять, сортировать записи и выполнять выборку с использованием фильтров.

#### Методические указания

Создайте заготовку программы MyDb, как указано в прил. 5.

Для отображения ресурсов приложения щелкните на корешке вкладки **Окно ресурсов**. Разверните дерево ресурсов, щелкнув на знаке «+» перед папкой MyDb.rc. Далее откройте папку ресурсов Dialog и сделайте двойной щелчок на идентификаторе диалогового окна IDD\_MYDB\_FORM и тем самым откройте диалог в редакторе ресурсов.

Пользуясь инструментами редактора диалогового окна, добавьте в него текстовые поля редактирования и статические надписи по образцу, показанному на рис. 4.1. Присвойте полям редактирования идентификаторы в соответствии с шаблоном: IDC\_названиетаблицы\_названиеполя (например, для поля ID таблицы User IDC\_UserID, а для поля UserName – идентификатор IDC\_UserName). Для текстового поля, содержащего идентификатор IDC\_UserID, установите свойство **Только для чтения** в True (определяется в окне свойств). 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 **Только для чтения** установлен для первого (текстового) поля по той причине, что оно будет содержать первичный ключ базы данных, который не подлежит изменению.

Напишите код, который связывает элемент редактирования IDC\_UserID с полями таблицы базы данных:

```
DDX_FieldText(pDX, IDC_UserID, m_pSet->m_id, m_pSet);
DDX_FieldText(pDX, IDC_UserSurname, m_pSet->m_surname, m_pSet);
DDX_FieldText(pDX, IDC_UserName, m_pSet->m_name, m_pSet);
DDX_FieldText(pDX, IDC_UserLastName, m_pSet->m_lastname, m_pSet);
DDX_FieldText(pDX, IDC_UserPsp, m_pSet->m_psp, m_pSet);
DDX_FieldText(pDX, IDC_UserVsrpp, m_pSet->m_vsrpp, m_pSet);
DDX_FieldText(pDX, IDC_UserPhysics, m_pSet->m_physics, m_pSet);
```

Откомпилируйте и запустите программу, и вы увидите окно, показанное на рис. 4.2. Приложение отображает содержимое записей таблицы user. Используя элементы навигации, расположенные на панели инструментов, можно перемещаться от одной записи таблицы user к другой. Чтобы обновить

любую из записей, достаточно просто изменить содержимое любого из полей записи. При переходе к другой записи приложение автоматически перенесет отредактированные данные в таблицу.

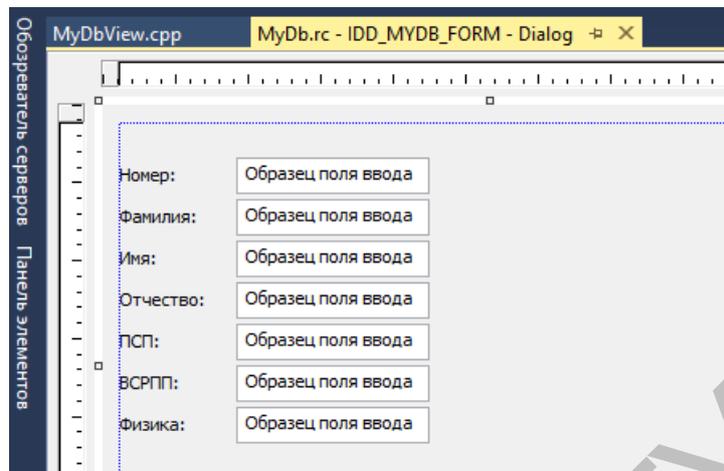


Рис. 4.1. Создание формы для базы данных

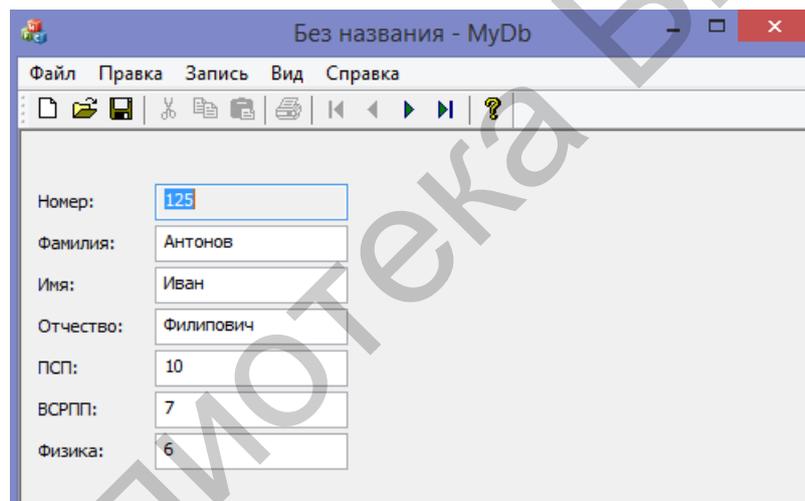


Рис. 4.2. Отображение в приложении данных из таблицы user

**Добавление и удаление записей.** Добавление и удаление записей в таблице базы данных реализуются достаточно просто благодаря существованию в Visual C++ классов `CRecordView` и `CRecordSet`, предоставляющих все необходимые методы. Для добавления к приложению команд *Добавить* и *Удалить* необходимо выполнить следующие действия:

1. Щелкните левой кнопкой мыши на корешке вкладки *Окно ресурсов*, откройте папку `Menu` и сделайте двойной щелчок левой кнопкой мыши на меню `IDR_MAINFRAME`. На экране раскроется окно редактора меню. Щелкните левой кнопкой мыши в меню *Запись* и тем самым откройте его, а затем щелкните левой кнопкой мыши на пустой области в нижней части этого меню. Вызовите свойства. В поле *ID* введите значение `ID_RECORD_ADD`, а в поле *Надпись*

введите значение **&Добавить запись**. В результате в меню **Запись** будет добавлена новая команда.

2. В следующий пустой элемент меню внесите команду удаления, имеющую идентификатор **ID\_RECORD\_DELETE** (поле **ИД**) и **Надпись &Удалить**, как показано на рис. 4.3.

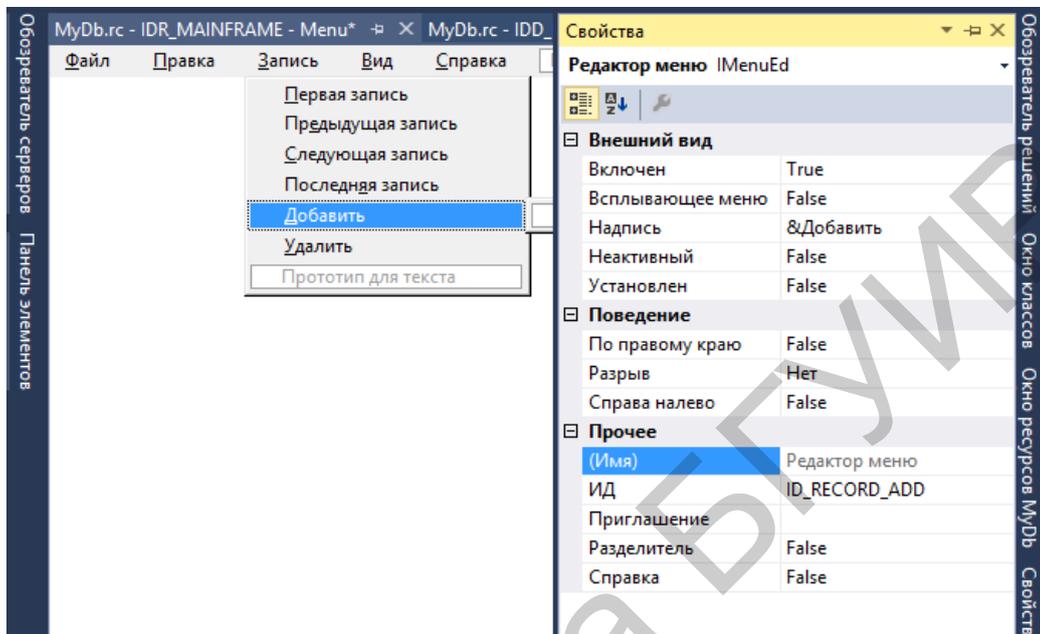


Рис. 4.3. Добавление в меню команд добавления и удаления записей

Далее необходимо добавить на панель инструментов пару новых пиктограмм и связать их с командами меню. Для этого в **Окне ресурсов** откройте папку **Toolbar** и сделайте двойной щелчок на идентификаторе **IDR\_MAINFRAME\_256**. Панель инструментов приложения будет отображена в окне редактора ресурсов.

Щелкнув левой кнопкой мыши на пустой пиктограмме панели инструментов, выберите ее, а затем с помощью инструментов графического редактора нарисуйте на ней голубой знак «+», как показано на рис. 4.4. Сделайте двойной щелчок левой кнопкой мыши на новой пиктограмме панели инструментов. Раскроется окно свойств. В списке **ИД** выберите значение **ID\_RECORD\_ADD**.

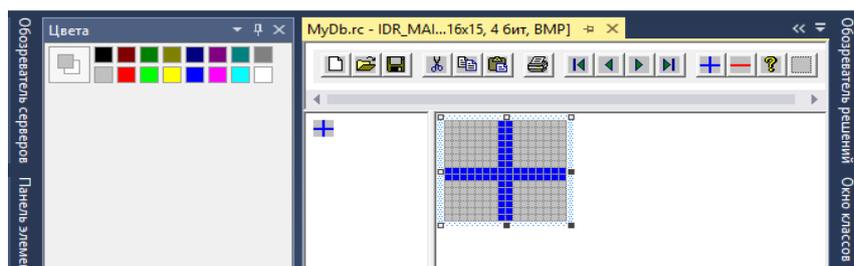


Рис. 4.4. Добавление пиктограмм на панель инструментов

Снова выделите пустую пиктограмму и нарисуйте на ней красный знак «←», присвойте пиктограмме идентификатор ID\_RECORD\_DELETE. Перетащите пиктограммы добавления и удаления левее пиктограммы справки, помеченной знаком вопроса.

Теперь, когда в меню уже добавлены новые команды и на панель инструментов помещены соответствующие пиктограммы, необходимо сформировать программный код, который будет обрабатывать командные сообщения, посылаемые, когда пользователь щелкает на пиктограмме или выбирает пункт меню. Так как в нашем приложении с базой данных связан класс представления, в нем и следует организовать перехват этих сообщений. Выполните следующие операции:

1. Выберите пункт меню *Добавить* и щелчком правой кнопки мыши откройте контекстное меню, в котором выберите *Добавить обработчик событий...* (рис. 4.5).

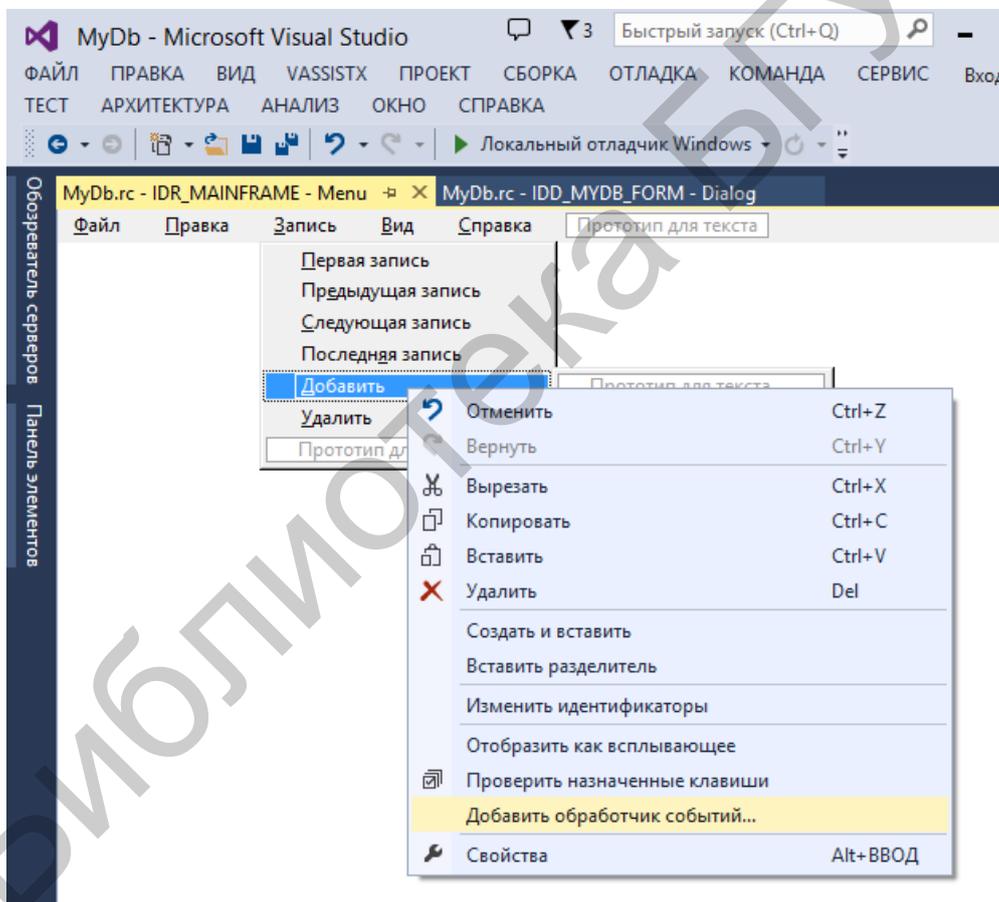


Рис. 4.5. Добавление обработчика события пункта меню *Добавить*

2. Слева выберите тип сообщения COMMAND, справа в списке выберите класс CMyDbView, в котором будет реализована функция для обработки пункта меню, снизу запишите имя функции – OnRecordAdd. Затем нажмите кнопку *Добавить=>Править* (рис. 4.6).

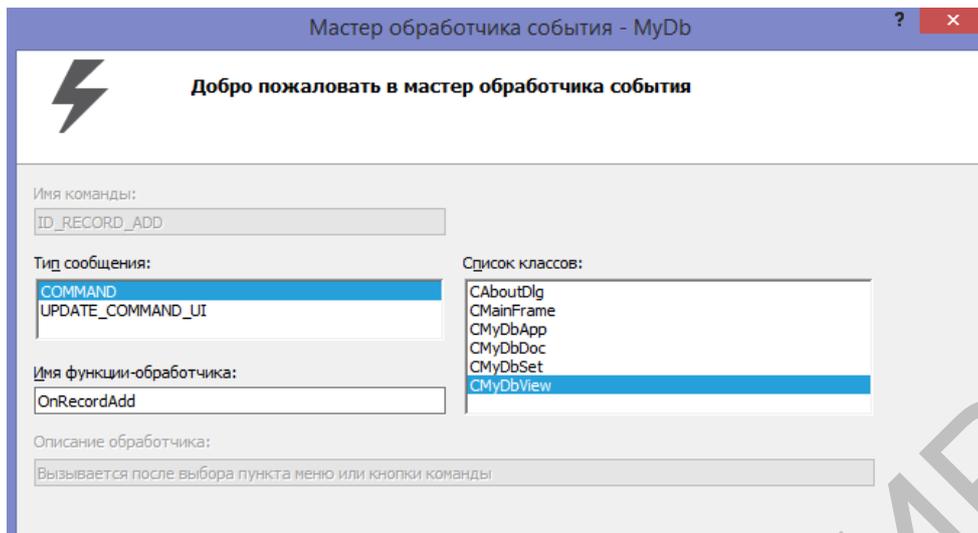


Рис. 4.6. Добавление обработчика события выбора пункта меню *Добавить*

Аналогичным образом добавим метод для обработки пункта меню *Удалить*.

В окне *Обозреватель решений* откройте файл MyDbView.h. В объявлении класса добавьте следующие строки в раздел Attributes:

```
protected:
BOOL m_bAdding;
```

В окне *Просмотр классов* сделайте двойной щелчок левой кнопкой мыши на конструкторе класса CMyDbView и добавьте следующую строку в конец этой функции:

```
m_bAdding = FALSE;
```

Сделайте двойной щелчок левой кнопкой мыши на функции OnRecordAdd() и отредактируйте ее текст так, как показано ниже.

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

В окне *Просмотр классов* щелкните правой кнопкой мыши на элементе CMyDbView и выберите в раскрывшемся контекстном меню команду *Добавить=>Добавить функцию....* В поле имени функции определите OnMove (рис. 4.7), выберите возвращаемое значение функции bool, тип

принимаемого параметра – `unsigned int`, имя принимаемого параметра – `nIDMoveCommand`, поставьте галочку возле **Виртуальная**. Затем щелкните на кнопке **Готово**. В результате в класс будет добавлена функция и можно будет немедленно отредактировать заготовку ее текста.

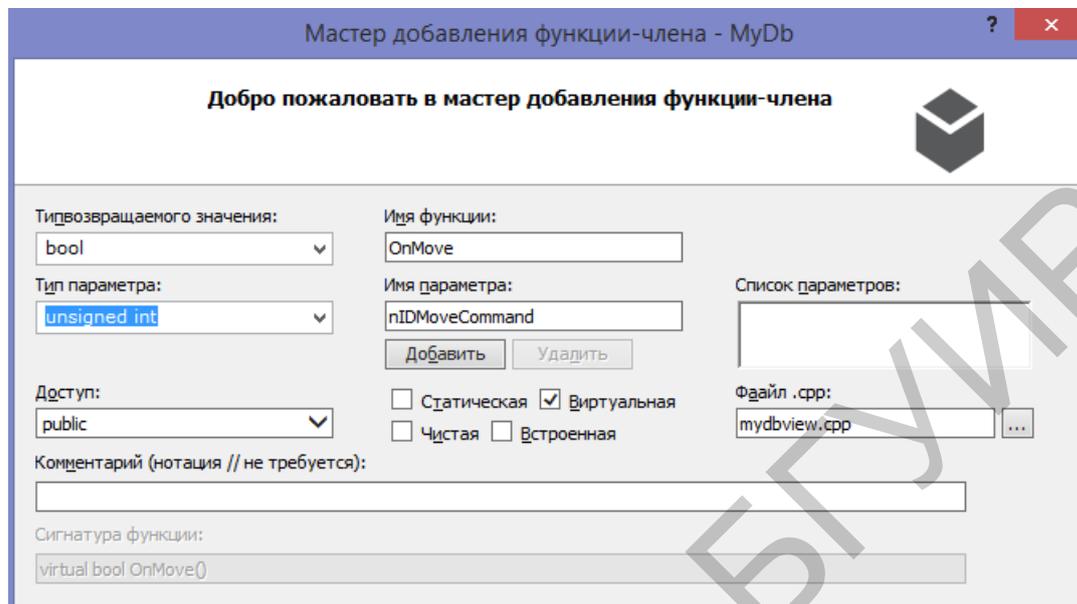


Рис. 4.7. Переопределение функции OnMove

Отредактируйте функцию `OnMove()` так, как показано ниже.

```

BOOL CMyDbView::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_UserID);
        pCtrl->SetReadOnly (TRUE);
        return TRUE;
    } else
        return CRecordView::OnMove (nIDMoveCommand);
}

```

Аналогично `OnRecordAdd()` добавьте `OnRecordDelete()` и отредактируйте ее следующим образом:

```

void CDBView::OnRecordDelete ()

```

```

{
    m_pSet->Delete();
    m_pSet->MoveNext();
    if (m_pSet->IsEOF())
        m_pSet->MoveLast();
    if (m_pSet->IsBOF())
        m_pSet->SetFieldNull(NULL);
    UpdateData(FALSE);
}

```

Модифицированное приложение готово, и теперь оно способно выполнять добавление и удаление записей, равно как и их обновление.

**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. Для добавления функции сортировки и фильтрации в наше приложение выполните следующие действия.

Добавьте меню **Сортировка** в основное меню приложения, как показано на рис. 4.8. Предоставьте Visual Studio автоматически определить идентификаторы команд.

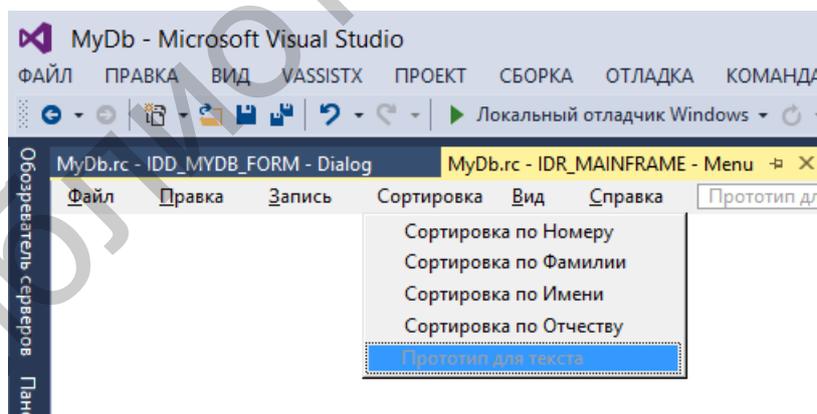


Рис. 4.8. Создание меню **Сортировка**

С помощью команды контекстного меню **Добавить обработчик событий...** организуйте в классе CMyDbView перехват четырех новых команд сортировки, задав для них соответствующие имена функций (рис. 4.9).

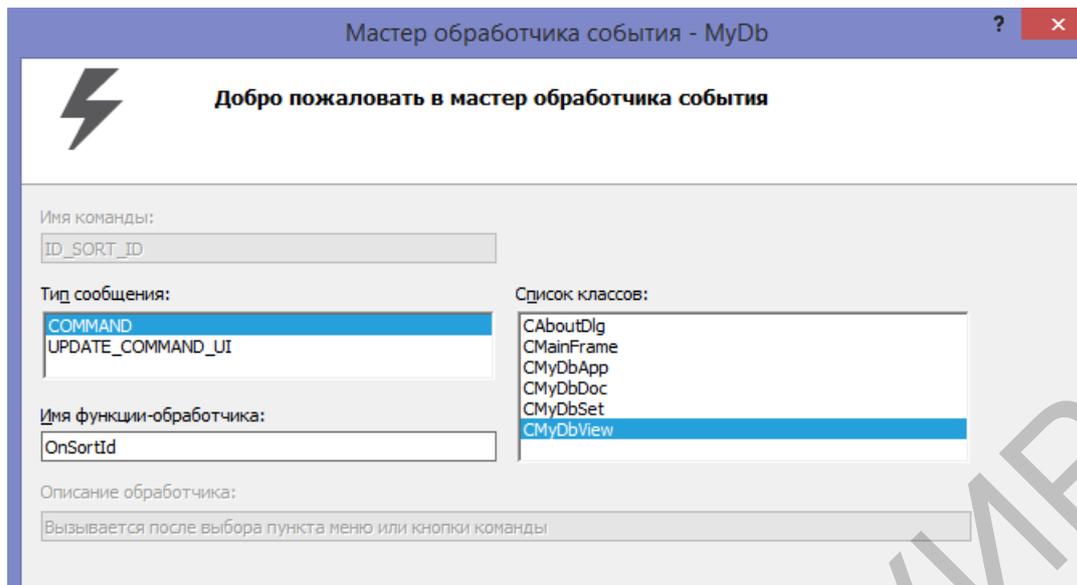


Рис. 4.9. Добавление функции сортировки

Добавьте меню **Фильтрация** в строку меню приложения. Предоставьте Visual Studio установить идентификаторы команд, как показано на рис. 4.10.

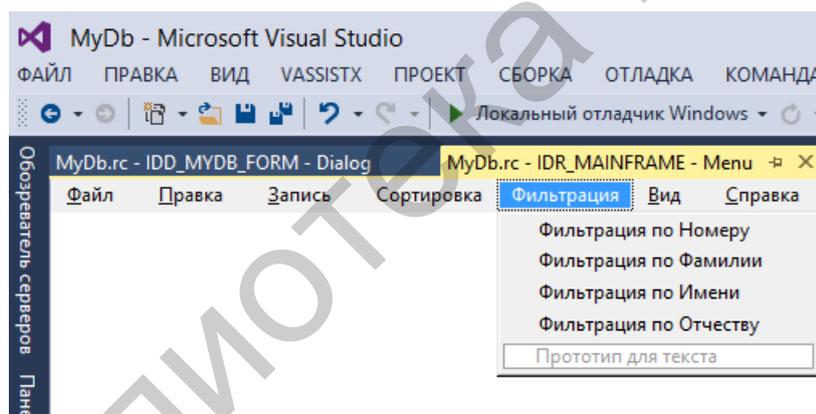


Рис. 4.10. Создание меню **Фильтрация**

С помощью команды контекстного меню **Добавить обработчик событий...** организуйте в классе CMyDbView перехват четырех новых команд сортировки, задав для них соответствующие имена функций (рис. 4.11).

Создайте новое диалоговое окно, выбрав в контекстном меню диалога **Вставить Dialog**, а затем отредактируйте его так, чтобы оно выглядело, как показано на рис. 4.12.

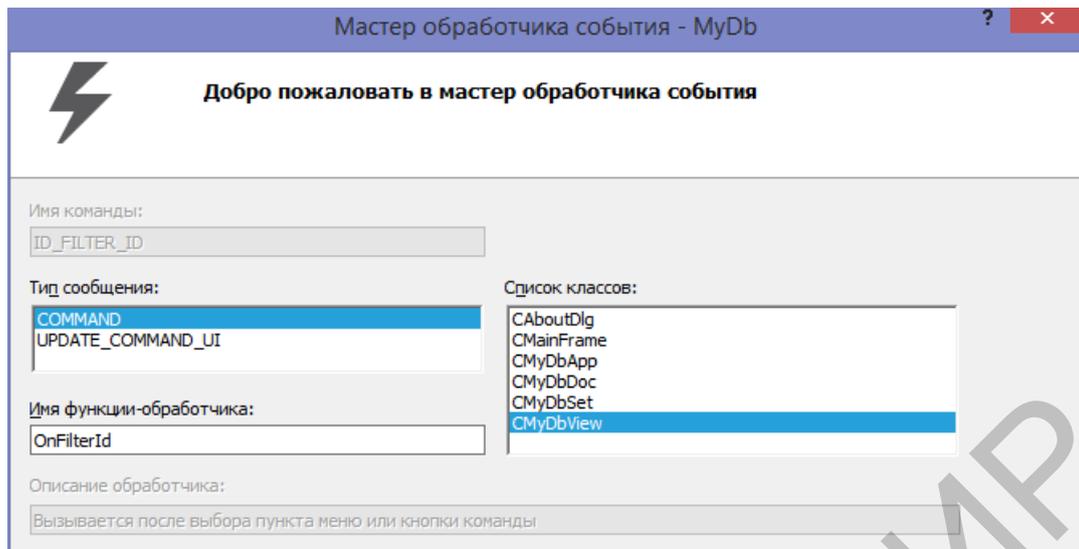


Рис. 4.11. Добавление функции фильтрации

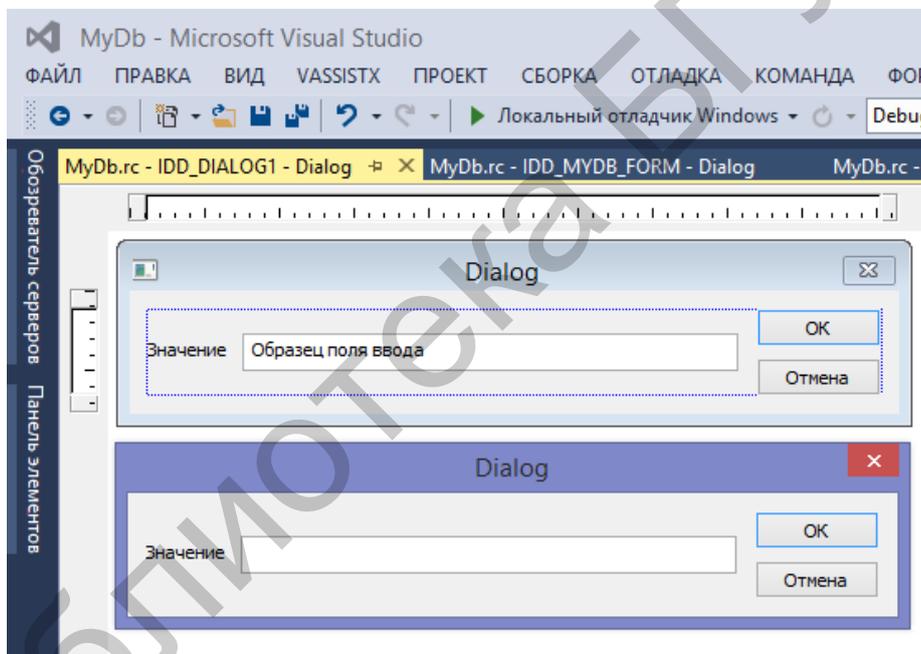


Рис. 4.12. Создание диалогового окна установки параметров фильтрации

Присвойте элементу управления – текстовому полю – идентификатор ID\_FILTERVALUE. Выбрав созданное диалоговое окно в контекстном меню, выберите *Добавить класс....* В поле *Имя класса* введите значение CFilterDlg. Свяжите элемент управления IDC\_FILTERVALUE с переменной-членом m\_filterValue с помощью *Добавить переменную....*

Отредактируйте текст функций, связанных с командами сортировки, в соответствии со следующим содержанием:

```
void CMyDbView::OnSortID()
{
```

```

    m_pSet->Close();
    m_pSet->m_strSort = "[id]";
    m_pSet->Open();
    UpdateData (FALSE);
}
void CMyDbView::OnSortSurname()
{
    m_pSet->Close();
    m_pSet->m_strSort = "[surname]";
    m_pSet->Open();
    UpdateData (FALSE);
}
void CMyDbView::OnSortName()
{
    m_pSet->Close();
    m_pSet->m_strSort = "[name]";
    m_pSet->Open();
    UpdateData (FALSE);
}
void CMyDbView::OnSortLastname()
{
    m_pSet->Close();
    m_pSet->m_strSort = "[lastname]";
    m_pSet->Open();
    UpdateData (FALSE);
}

```

Введите в начало файла MyDbView.cpp после уже имеющихся директив #include строку

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

**Добавьте функции-обработчики:**

```

void CMyDbView::OnFilterID()
{
    DoFilter((CString)"[id]", 1);
}
void CMyDbView::OnFilterSurname()
{
    DoFilter((CString)"[surname]", 0);
}
void CMyDbView::OnFilterName()
{
    DoFilter((CString)"[name]", 0);
}
void CMyDbView::OnFilterLastname()
{
    DoFilter((CString)"[lastname]", 0);
}

```

Далее необходимо будет написать функцию, выполняющую фильтрацию записей базы данных, представленных в классе выборки данных. На панели **Обозреватель классов** щелкните правой кнопкой мыши на классе CMyDbView и выберите в раскрывшемся контекстном меню команду **Добавить=>Добавить функцию....** Укажите в раскрывшемся диалоговом окне тип функции void и введите ее объявление как DoFilter(CString col). Данный метод должен быть защищенным, т. к. он вызывается только другими методами этого же класса CMyDbView. На панели **Обозреватель классов** сделайте двойной щелчок на функции DoFilter() и поместите в нее текст программы:

```
void CMyDbView::DoFilter(CString col, bool flag)
{
    CFilterDlg dlg;
    CString str;
    if (dlg.DoModal() == IDOK)
        if (flag)
            str = col + (CString)" = " + dlg.m_filterValue;
        else
str = col + (CString)" = '" + dlg.m_filterValue + (CString)''";
        m_pSet->m_strFilter = str;
        m_pSet->Requery();
        int recCount = m_pSet->GetRecordCount();
        if (recCount == 0)
        {
            MessageBox((CString)"No matching records.");
            m_pSet->m_strFilter = "";
            m_pSet->Requery();
        }
        m_pSet->MoveFirst();
        UpdateData(FALSE);
}
```

Оттранслируйте приложение и запустите его на выполнение. На экране появится главное окно приложения, теперь можно сортировать записи по любому полю, для чего достаточно просто выбрать имя поля в меню **Сортировка**. Кроме того, появилась возможность задать фильтрацию отображаемых записей, выбрав имя требуемого поля в меню **Фильтрация**, а затем введя значение фильтра в раскрывшемся диалоговом окне.

**Анализ функции OnSortName().** Все функции сортировки имеют одинаковую структуру. Они закрывают выборку данных, устанавливают свои переменные-члены m\_strSort в выборке и снова открывают выборку данных, а затем вызывают функцию UpdateData() для обновления окна представления данными из вновь полученной отсортированной выборки данных. Объект класса CRecordSet использует специальную строковую переменную m\_strSort для определения способа упорядочения записей.

**Анализ функции DoFilter().** 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. Для выполнения фильтрации выборка данных должна быть закрыта, а затем при ее повторном открытии, функция `DoFilter()` выполнит формирование выборки данных с учетом требуемой фильтрации.

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()`.

### Задания

1. Разработать приложение управления базой данных сдачи студентами экзаменационной сессии.
2. Разработать приложение управления базой данных студенческого общежития.
3. Разработать приложение управления базой данных выданных студентам курсовых.
4. Разработать приложение управления базой данных посещения студентами занятий.
5. Разработать приложение управления базой данных студенческой библиотеки.
6. Разработать приложение управления базой данных студенческой бухгалтерии для выдачи стипендий.
7. Разработать приложение управления базой данных распределения путевок студенческого профкома.
8. Разработать приложение управления базой данных, содержащей расписание студенческих занятий.
9. Разработать приложение управления базой данных коммунальных платежей за квартиру.
10. Разработать приложение управления базой данных учета времени доступа к Интернету.
11. Разработать приложение управления базой данных учета времени переговоров по телефону.

12. Разработать приложение управления базой данных учета страховых полисов.

В разрабатываемом приложении обеспечить добавление, редактирование и удаление записей из базы данных, сохранение результатов в файле (создание текстового отчета), а также предусмотреть возможность возвращения к просмотру всех записей базы данных после выполнения фильтрации.

Библиотека БГУИР

## ЛАБОРАТОРНАЯ РАБОТА №5 ИСПОЛЬЗОВАНИЕ ТЕХНОЛОГИИ OLE DB

Цель работы – ознакомиться с процессом создания приложения доступа к источнику данных с использованием технологии OLE DB.

### Методические указания

Создайте заготовку программы по шаблону, указанному в прил. 5, с небольшими изменениями. Для выбора OLE DB в качестве объекта доступа к данным на шаге 2 в диалоге **Поддержка базы данных** (Database options) в группе переключателей **Тип клиента** (Data Source) выберите переключатель OLE DB, после чего кнопка с надписью **Источник данных** (Select OLE DB Data source) станет активной. Нажмите на эту кнопку. В результате появится окно **Свойство канала передачи данных** (Data Link Properties) (рис. 5.1).

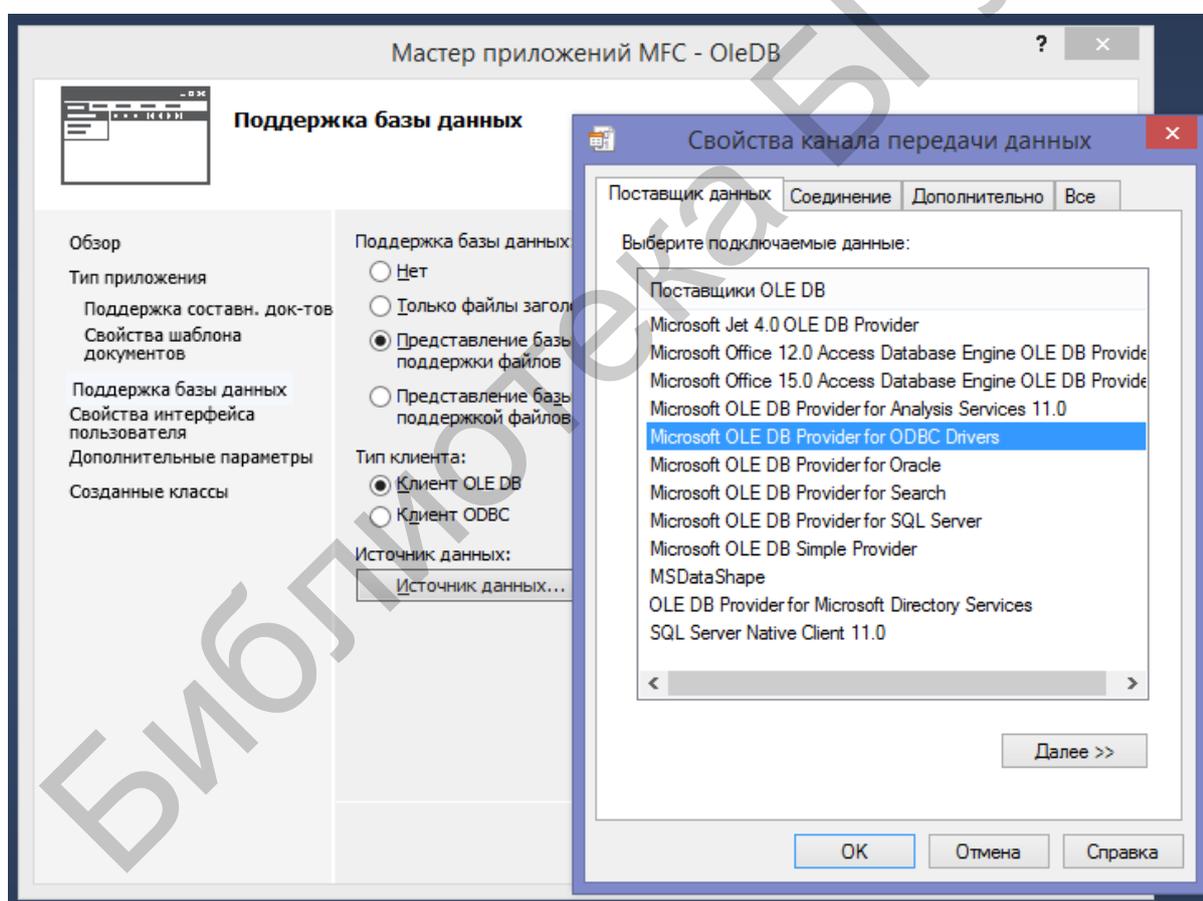


Рис. 5.1. Диалог выбора провайдера ODBC драйвера

Затем нажмите кнопку **Далее**, что переводит нас на закладку **Соединение**. Теперь в элементе **Использовать имя источника данных** выберите источник

данных (в нашем примере это база данных с именем test\_database\_laba5.mdb) (рис. 5.2).

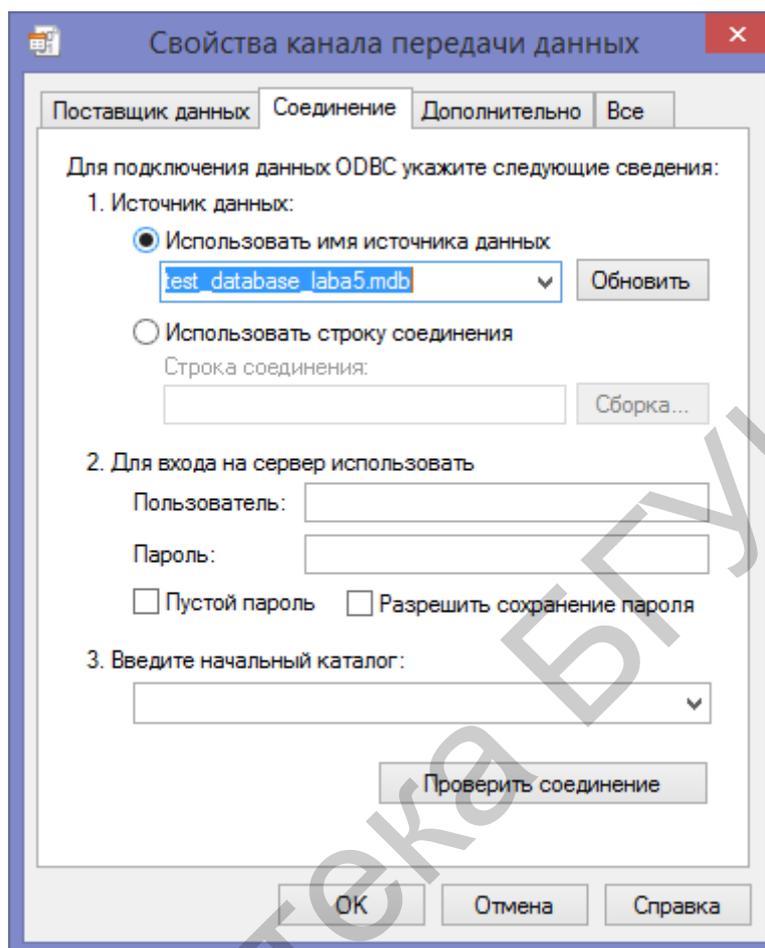


Рис. 5.2. Выбор источника данных

После этого в строке 3. **Введите начальный каталог** выберите абсолютный путь к источнику данных. Затем нажмите кнопку **Проверить соединение**. При успешном соединении с источником данных на экран выводится сообщение «Проверка соединения выполнена» (рис. 5.3).

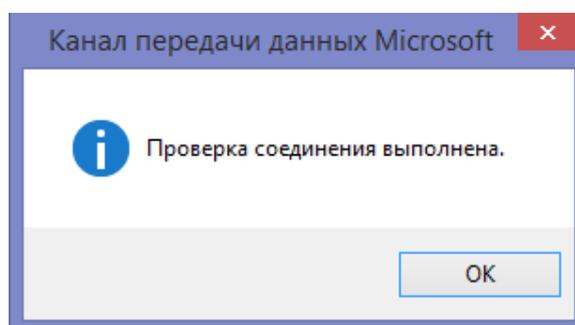


Рис. 5.3. Сообщение об успешном тестировании подсоединения к источнику данных

Для разграничения доступа при обращении к источнику данных необходимо перейти на закладку *Дополнительно*, где в графе *Права доступа* следует выбрать соответствующие права доступа. В нашем случае выберите ReadWrite (*Чтение/Запись*), как показано на рис. 5.4.

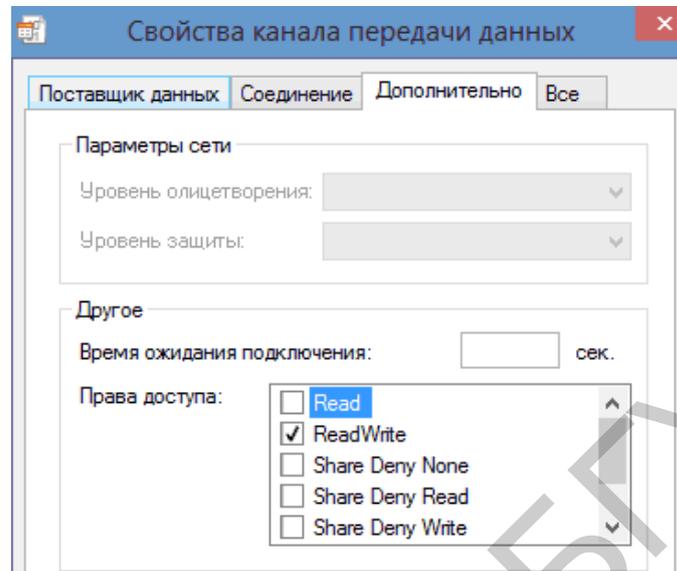


Рис. 5.4. Назначение прав доступа на работу с источником данных

После нажатия кнопки ОК на экране появляется диалог *Введите объект базы данных*, в котором выберите имя таблицы, с которой хотите работать (в нашем примере это user) (рис. 5.5).

Последующие шаги можно пропустить, нажав кнопку *Готово*.

После этого *Мастер приложений* автоматически генерирует классы, содержащие методы, с помощью которых мы можем работать с нашей базой данных.

Теперь займитесь построением графического интерфейса. Расположите элементы управления типа Edit Control и Static Text так, как показано на рис. 5.6. И сразу же проставьте для элементов соответствующие ИД (IDC\_EDIT\_ID, IDC\_EDIT\_SURNAME, IDC\_EDIT\_NAME, IDC\_EDIT\_LASTNAME).

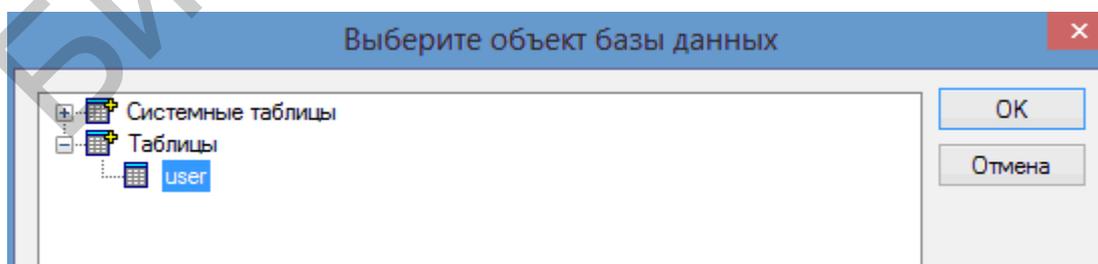


Рис. 5.5. Диалог выбора таблицы в источнике данных

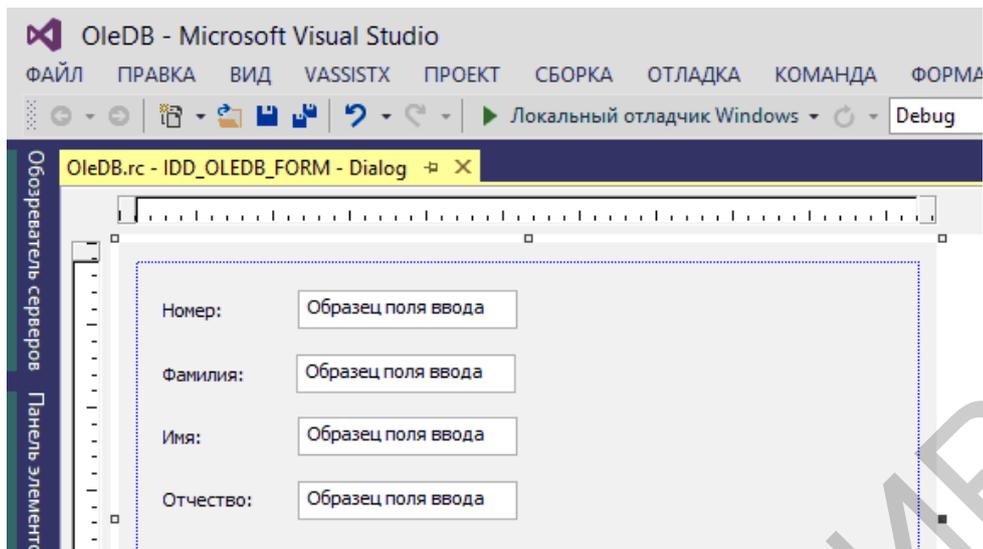


Рис. 5.6. Расположение элементов управления на форме

Теперь для позиционирования курсора в источнике данных в классе COleDBView создайте метод displayMembersOfDataSource() следующим образом.

```
void COleDBView::displayMembersOfDataSource()
{
  CString str;
  str.Format(_T("%d"), m_pSet->m_id); //форматирование
  m_edit_id.SetWindowText(str); // визуальное отображение данных
  str = m_pSet->m_surname;
  m_edit_surname.SetWindowText(str);
  str = m_pSet->m_name;
  m_edit_name.SetWindowText(str);
  str = m_pSet->m_lastname;
  m_edit_lastname.SetWindowText(str);
}
```

Функция Format() позволяет преобразовать данные разных типов в объект класса CString. В нашем случае это позволит преобразовать целочисленные данные из источника данных в строковые. Функция SetWindowText() выводит в элементы EditControl содержимое текущей позиции курсора источника данных. Переменная m\_pSet является указателем, генерируемым **Мастером приложений**, на объект класса COleDBSet, через который реализуется связь нашего приложения с базой данных.

Теперь в **Мастере классов** создайте переменные для наших элементов управления типа Edit Control (рис. 5.7).

Перейдите к описанию событий для работы с курсором источника данных. Для этого необходимо переопределить стандартные методы, используемые элементами навигации, расположенными в элементе управления Toolbar.

Откройте окно **Свойства** класса `OleDbView`, перейдите к обработчику событий ⚡, найдите группу идентификаторов, отвечающих за навигацию по записям источника данных (`ID_RECORD_FIRST`, `ID_RECORD_LAST`, `ID_RECORD_NEXT`, `ID_RECORD_PREV`), и создайте обработчики (рис. 5.8).

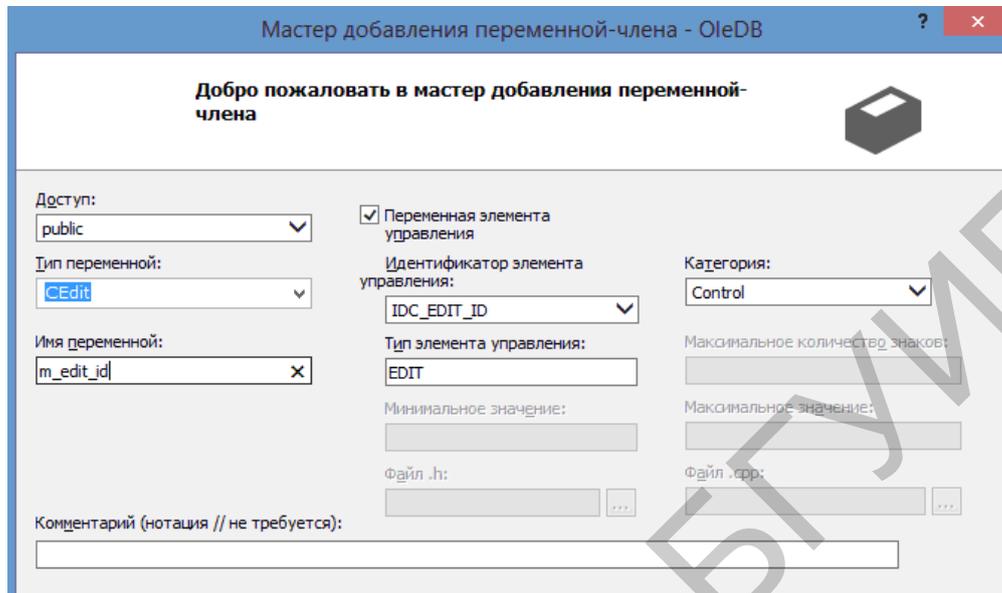


Рис. 5.7. Окно добавления переменной

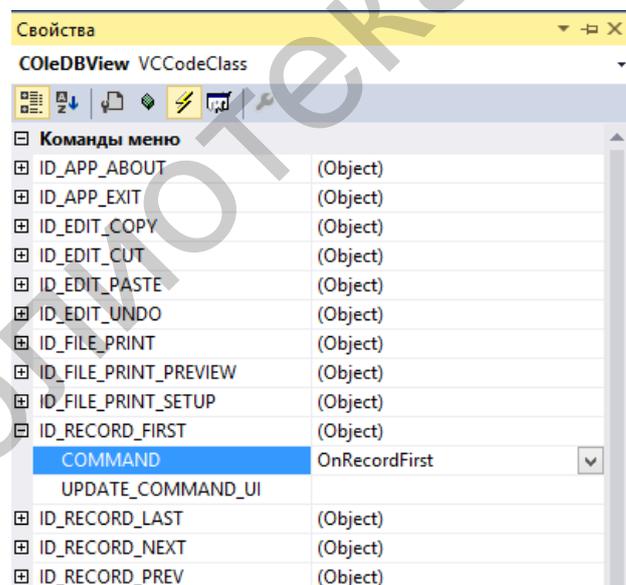


Рис. 5.8. Диалог определения имени метода `OnRecordFirst`

Отредактируйте созданные методы, как показано ниже.

```
void COleDbView::OnRecordFirst()
{
    m_pSet->MoveFirst(); // переход на первую позицию
    displayMembersOfDataSource();
}
```

```

}
void COleDBView::OnRecordLast()
{
    m_pSet->MoveLast();
    displayMembersOfDataSource();
}
void COleDBView::OnRecordNext()
{
    m_pSet->MoveNext();
    displayMembersOfDataSource();
}
void COleDBView::OnRecordPrev()
{
    m_pSet->MovePrev();
    displayMembersOfDataSource();
}

```

Функция `MoveFirst()` перемещает курсор источника данных на первую позицию (т. е. на первую запись), а следующий за ней вызов метода `displayMembersOfDataSource()` приводит к отображению данных в элементах Edit Control.

*Замечание.* Для того чтобы снять блокировку с кнопок навигации к первой и предыдущей записям (`ID_RECORD_FIRST`, `ID_RECORD_PREV`), необходимо последовательно выбрать эти идентификаторы, создавая на каждый из них обработчик типа `UPDATE_COMMAND_UI`.

Перейдите к созданию механизмов удаления, редактирования и добавления информации из источника данных.

Прежде чем создавать методы добавления и изменения данных в источнике, необходимо в классе `COleDBView` создать метод `COleDBView::FillOleDBParameters()`, который инициализирует переменные класса значениями из элементов типа Edit Control, для последующего добавления и редактирования.

```

void COleDBView::FillOleDBParameters()
{
    int i;
    CString str;
    char chars[40]; // объявление массива типа char
    m_edit_id.GetWindowTextW(str);
    // инициализация переменной m_id_table
    m_pSet->m_id = _ttoi(str);
    m_edit_name.GetWindowTextW(str);
    // выделение необходимой памяти
    memset(chars, 0, 40);
    for (i = 0; i < str.GetLength(); i++)
        // инициализация массива chars
        chars[i] = (unsigned char)str[i];
    // инициализация переменной m_surname
    strcpy(m_pSet->m_surname, chars);
    // обновление блока выделенной памяти
}

```

```

memcpy(m_pSet->m_surname, &chars[0], 40);
m_edit_name.GetWindowTextW(str);
memset(chars, 0, 40);
for (i = 0; i < str.GetLength(); i++)
    chars[i] = (unsigned char)str[i];
strcpy(m_pSet->m_name, chars);
memcpy(m_pSet->m_name, &chars[0], 40);
m_edit_lastname.GetWindowTextW(str);
memset(chars, 0, 40);
for (i = 0; i < str.GetLength(); i++)
    chars[i] = (unsigned char)str[i];
strcpy(m_pSet->m_lastname, chars);
memcpy(m_pSet->m_lastname, &chars[0], 40);
}

```

Метод `GetWindowTextW()` возвращает содержимое элементов типа `EditBox`. Для того чтобы привести полученные данные к нужному типу (`int` и `char*`), выделим для переменной `chars` необходимый размер памяти при помощи функции `memset()`. Следующий за ним цикл выполняет посимвольное явное преобразование типов. Функция `strcpy()` копирует данные из переменной `chars` в переменную `m_surname`. Для того чтобы в дальнейшем результаты копирования можно было использовать, необходимо произвести обновление значений переменных. Это достигается использованием функции `memcpy()`.

Так как метод `COleDBView::FillOleDBParameters()` инициализирует переменные таблицы типом `char`, то в классе `COleDBSet` следует изменить типы данных, которые отвечают за связь с базой данных в соответствии с методом `COleDBView::FillOleDBParameters()`:

```

class COleDBSetAccessor
{
public:
    int m_id;
    char m_surname[256];
    char m_name[256];
    char m_lastname[256];
    ...
}

```

**Добавление данных.** Добавьте в класс `COleBDView` переменную `m_bAdding` типа `bool` и инициализируйте ее в конструкторе класса значением `false`.

```

COleDBView::COleDBView()
: COleDBRecordView(COleDBView::IDD)
, m_bAdding(false)
, m_order_str(_T("id"))
, m_where_str(_T(""))

```

Создайте три новые кнопки: *Добавить*, *Удалить* и *Редактировать* (сразу проставьте ИД IDC\_INSERT, IDC\_EDIT, IDC\_DELETE), а также обработчики событий нажатия кнопок. Например, двойной щелчок левой кнопкой мыши на кнопке *Добавить* приведет к созданию обработчика OnBnClickedInsert() (сделайте то же для остальных кнопок). После этого в тело метода OnBnClickedInsert() добавьте следующий код:

```
void COleDBView::OnBnClickedInsert()
{
    if (!m_bAdding)
    {
        m_bAdding = true;
        SetDlgItemText(IDC_INSERT, _T("Сохранить"));
        GetDlgItem(IDC_EDIT_ID)->EnableWindow(SW_SHOW);
        SetDlgItemText(IDC_EDIT_ID, NULL);
        SetDlgItemText(IDC_EDIT_SURNAME, NULL);
        SetDlgItemText(IDC_EDIT_NAME, NULL);
        SetDlgItemText(IDC_EDIT_LASTNAME, NULL);
        GetDlgItem(IDC_EDIT_ID)->SetFocus();
    }
    else
    {
        m_pSet->MoveLast();
        FillOleDBParameters();
        m_pSet->Insert();
        m_pSet->MoveLast();
        SetDlgItemText(IDC_INSERT, _T("Добавить"));
        GetDlgItem(IDC_EDIT_ID)->EnableWindow(SW_HIDE);
        m_bAdding = false;
    }
}
```

Для кнопок редактирования и удаления добавьте следующий код:

```
void COleDBView::OnBnClickedEdit()
{
    FillOleDBParameters();
    m_pSet->SetData(); // изменение значений полей источника данных
    // передача источнику данных информации об изменениях,
    // выполненных над строкой.
    m_pSet->Update();
}
```

После инициализации переменных необходимо передать их новые значения источнику данных. Это достигается путем использования функции SetData(). Последующий вызов функции Update() приводит к передаче информации об изменениях в источнике данных.

```
void COleDBView::OnBnClickedDelete()
```

```

{
    m_pSet->Delete();
}

```

Для удаления записи вызывается метод Delete().

После этого для корректного добавления, вставки и удаления в класс COleDBSetAccessor необходимо добавить следующие свойства:

```

void GetRowsetProperties(CDBPropSet* pPropSet)
{
    pPropSet->AddProperty(DBPROP_CANFETCHBACKWARDS, true,
DBPROPOPTIONS_OPTIONAL);
    pPropSet->AddProperty(DBPROP_CANSROLLBACKWARDS, true,
DBPROPOPTIONS_OPTIONAL);
    pPropSet->AddProperty(DBPROP_IRowsetChange, true);
    pPropSet->AddProperty(DBPROP_IRowsetScroll, true);
    pPropSet->AddProperty(DBPROP_UPDATABILITY, DBPROPVAL_UP_CHANGE
| DBPROPVAL_UP_INSERT | DBPROPVAL_UP_DELETE);
}

```

При компиляции возможна ошибка:

```

error C1189: #error: Проблема безопасности: строка подключения
может содержать пароль

```

Для ее устранения в классе COleDBSetAccessor необходимо закомментировать строку:

```

// #error Проблема безопасности: строка подключения может
содержать пароль

```

На рис. 5.9 виден результат выполнения программы. На этом этапе изменения на форме сохраняются в базу, можно удалять и добавлять записи.

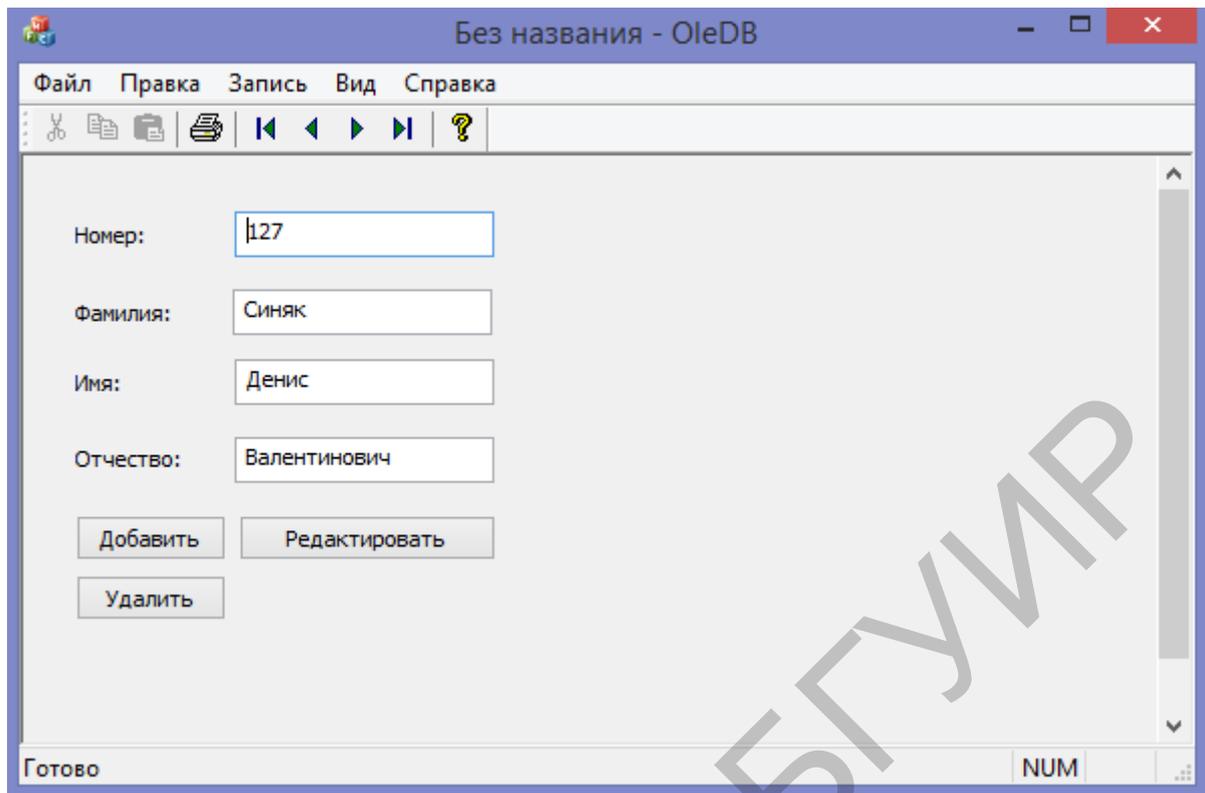


Рис. 5.9. Результат выполнения программы

Есть два класса: CTable и CCommand. Если нужно просто отобразить содержимое базы без сортировки и фильтрации, то используем CTable, если нужно большее – используем CCommand.

Если нужен класс CCommand, то откройте OleDbView.h и замените

```
class COleDBSet : public CTable<CAccessor<COleDBSetAccessor> >
```

на

```
class COleDBSet : public CCommand<CAccessor<COleDBSetAccessor> >
```

Там же в методе OpenRowset поменяйте строку

```
HRESULT hr = Open(m_session, L"users", pPropSet);
```

на

```
HRESULT hr = Open(m_session, L"select * from users", pPropSet);
```

Добавьте метод ReOpen.

```
HRESULT ReOpen(CString where_str = L"", CString order_str = L"")
{
```

```

CDBPropSet propset(DBPROPSET_ROWSET);
propset.AddProperty(DBPROP_IRowsetLocate, true);
GetRowsetProperties(&propset);
CString select_str;
if (where_str.IsEmpty())
select_str.Format(L"select * from users order by %s", order_str);
else
select_str.Format(L"select * from users where %s order by %s",
where_str, order_str);
Close();
HRESULT hr = Open(m_session, select_str, &propset);
#ifdef _DEBUG
if (FAILED(hr))
AtlTraceErrorRecords(hr);
#endif
return hr;
}

```

**Сортировка.** Добавьте два объекта типа CString в класс COleDBView: m\_where\_str и m\_order\_str.

Добавьте пункт меню для сортировки по номеру (рис. 5.10).

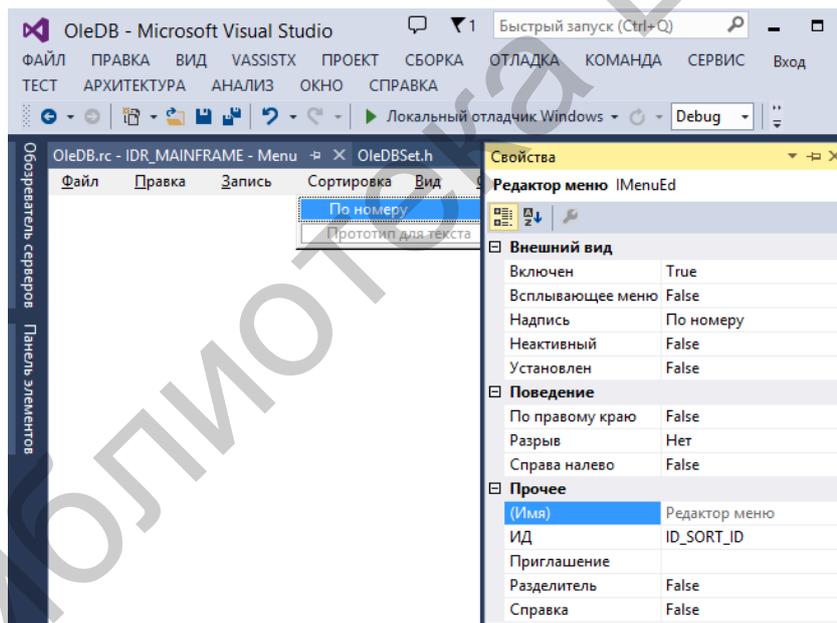


Рис. 5.10. Пункт меню сортировки по номеру

Создайте обработчик события (рис. 5.11).

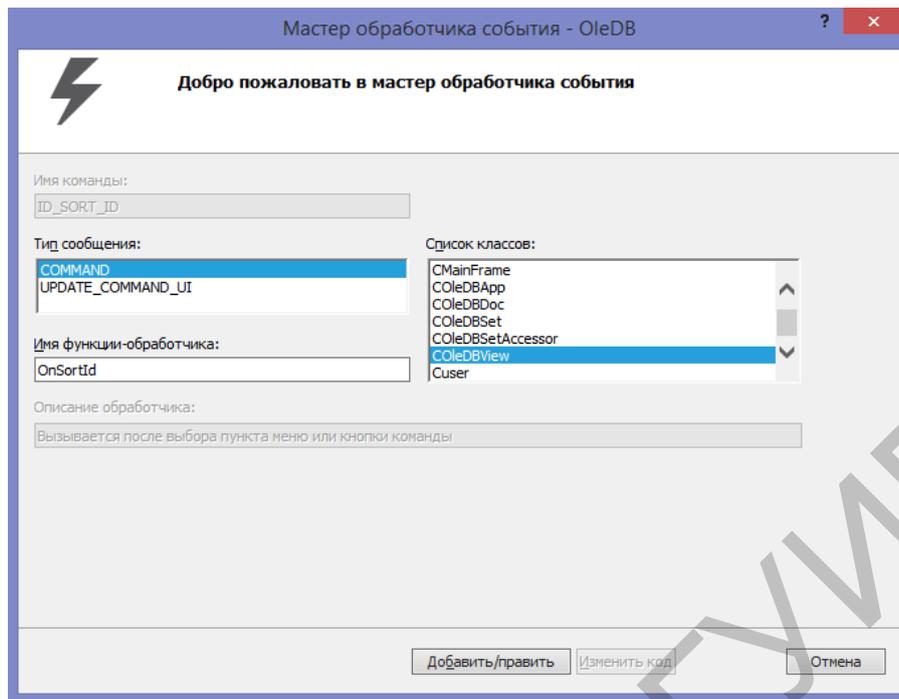


Рис. 5.11. Обработчик события *Сортировка по номеру*

Добавьте в него следующий код:

```
void COleDBView::OnSortId()
{
    m_order_str = L"id";
    ReQuery();
}
```

Для поддержки сортировки и фильтрации объявите новые переменные и методы в файле OleDBView.h:

```
public:
    CString m_order_str;
    CString m_where_str;
    void ReQuery(void);
    void DoFilter(CString column);
```

Инициализируйте переменные в конструкторе класса COleDBView:

```
COleDBView::COleDBView()
: COleDBRecordView(COleDBView::IDD)
, m_bAdding(false)
, m_order_str(_T("id"))
, m_where_str(_T(""))
```

Теперь создайте функцию ReQuery() (и прототип в OleDBView.h) в классе COleDBView со следующим кодом:

```

void COleDBView::ReQuery()
{
    m_pSet->ReOpen(m_where_str, m_order_str);
    OnMove(ID_RECORD_FIRST);
}

```

**Фильтрация.** Для фильтрации создайте пункт меню, как и в случае с сортировкой, и обработчик соответствующего события, код которого

```

void COleDBView::OnFilterId()
{
    DoFilter(L"id");
}

```

В класс COleBDView добавьте функцию DoFilter() со следующим кодом:

```

void COleDBView::DoFilter(CString column)
{
    CFilterDlg dlg;
    if (dlg.DoModal() == IDOK)
    {
        dlg.m_filterValue.Replace(L"\"", L"\""); // Замена на "\""
        dlg.m_filterValue.Replace(L"--", L"");
        m_where_str.Format(L"%s = '%s'", column, dlg.m_filterValue);
        ReQuery();
    }
}

```

Далее в ресурсах создайте новое диалоговое окно (рис. 5.12).

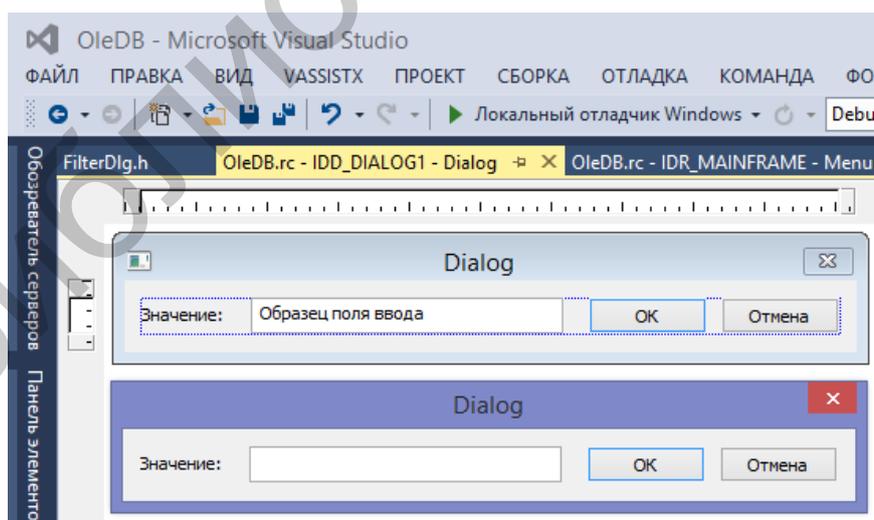


Рис. 5.12. Диалоговое окно фильтра по номеру

Создайте для нового диалогового окна класс CFilterDlg, в который добавьте Edit Control (с типом valuem\_filterValue) (рис. 5.13).

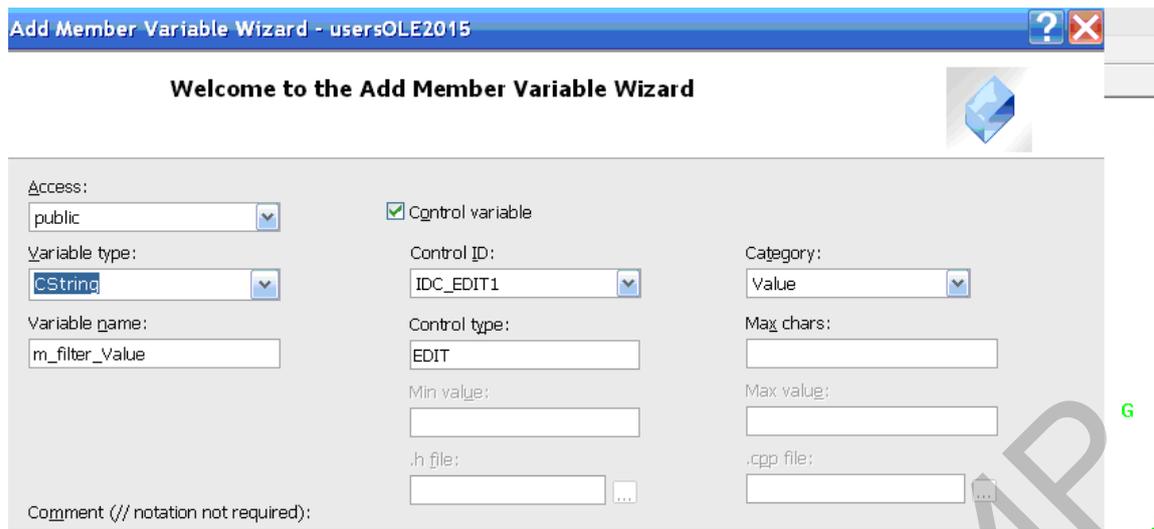


Рис. 5.13. Определение переменной m\_filterValue

В файл OleDbView.cpp после основных include добавьте:

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

Результат работы фильтра приведен на рис. 5.14.

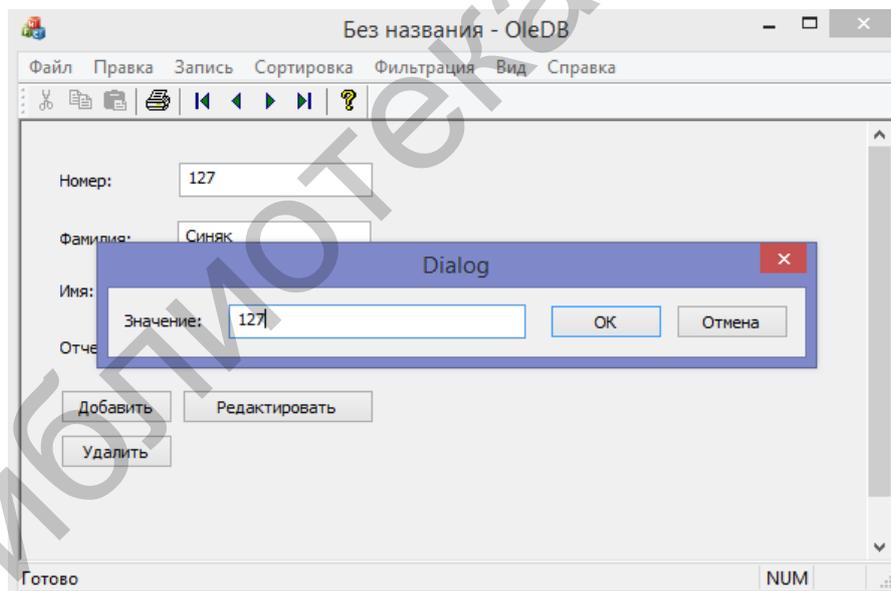


Рис. 5.14. Результат работы фильтра

### Задание

Разработать приложение для источника данных, использованного в лабораторной работе №4.

## ЛАБОРАТОРНАЯ РАБОТА №6

### ПРИМЕНЕНИЕ ТЕХНОЛОГИИ ADO ДЛЯ ДОСТУПА К БАЗЕ ДАННЫХ

Цель работы – ознакомиться с применением технологии ADO для доступа к базе данных, узнать назначение и возможности использования компонентов ActiveX.

#### Методические указания

Для ознакомления с технологией ADO необходимо разработать приложение управления базой данных учета лекарств в аптеке. В разрабатываемом приложении необходимо обеспечить добавление, редактирование, удаление, сортировку и фильтрацию записей таблицы. В качестве базы данных используйте MS Access, количество полей в таблице должно быть не менее четырех, типы полей – разные. Доступ к базам данных должен осуществляться через ADO.

В MS Access создайте базу данных аптека со следующими полями (рис. 6.1).

	Имя поля	Тип данных
🔑	id	Счетчик
	lekarstvo	Длинный текст
	nazn	Длинный текст
	recept	Длинный текст
	kol	Длинный текст
	cena	Денежный
	proizv	Длинный текст

Рис. 6.1. Поля базы данных аптека

В Microsoft Visual Studio создайте проект MFC на базе диалогового окна под именем ADO.

Добавьте ActiveX-элементы. Для этого необходимо в панели элементов вызвать контекстное меню щелчком правой кнопки мыши и выбрать пункт **Выбрать элементы...**, в ответ откроется диалоговое окно **Выбор элементов панели элементов**, в котором необходимо открыть вкладку **COM-компоненты** и поставить галочки напротив Microsoft ADO Data Control, version 6.0 (OLEDB), и Microsoft DataGrid Control 6.0 (SP6) (OLEDB) (рис. 6.2) .

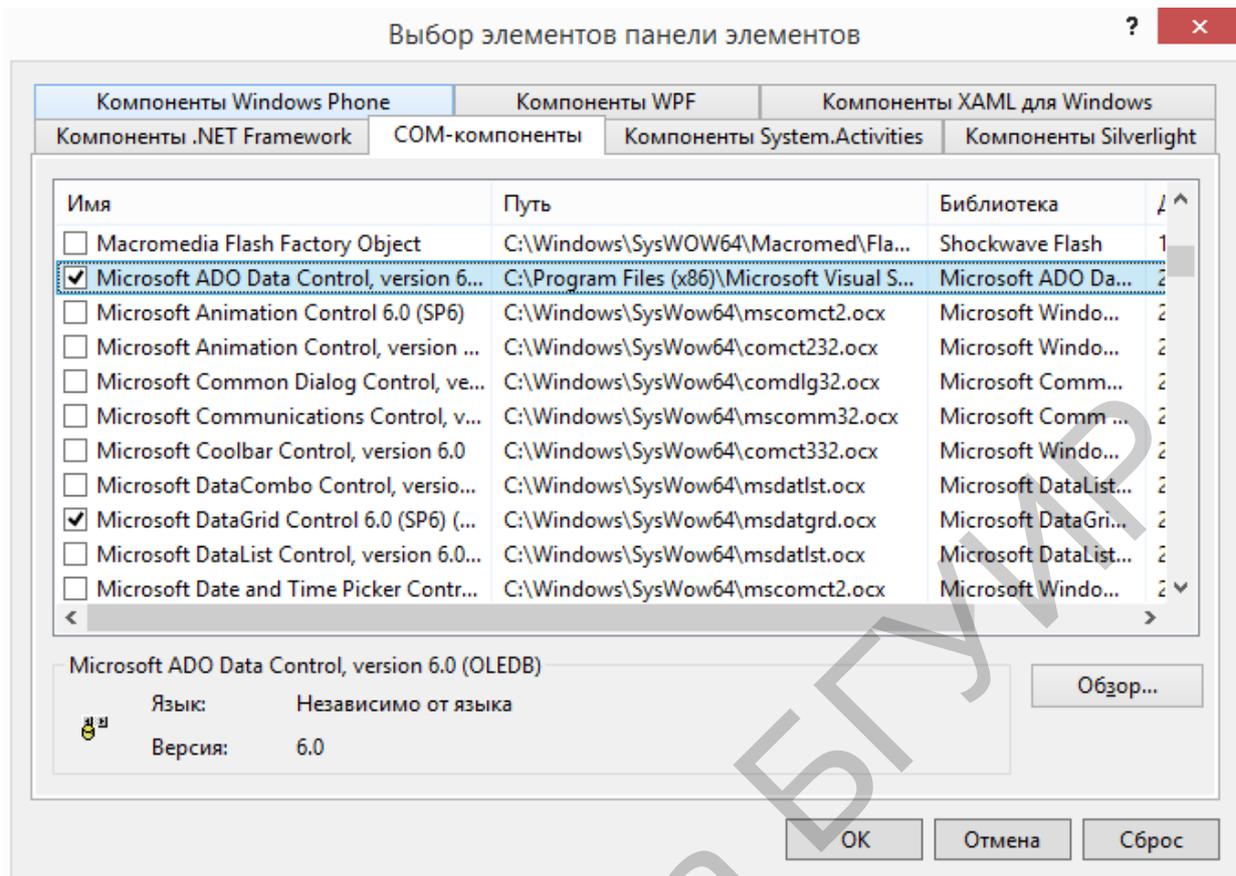


Рис. 6.2. Добавление ActiveX-элементов

Теперь на панели инструментов появились значки для ADO Data Control и DataGrid Control (рис. 6.3).

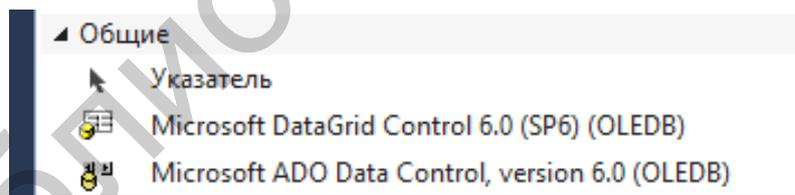


Рис. 6.3. Значки для ActiveX-элементов

Отредактируйте шаблон диалога, добавив в него два элемента Group Box, два элемента Combo Box, элемент Edit Control, элемент Check Box, ADO Data Control, DataGrid Control и элемент Button (рис. 6.4).

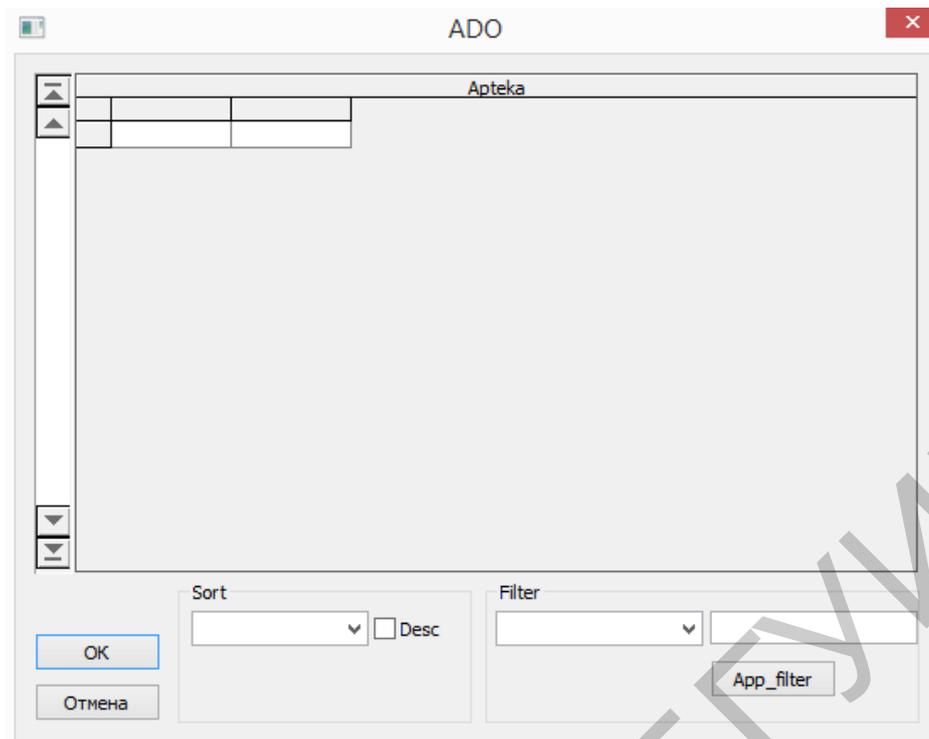


Рис. 6.4. Редактирование шаблона диалогового окна

Теперь подключите ADO Data Control к источнику данных.

В свойствах ADO Data Control выберите `ConnectionString=>Use Connection String` и нажмите Build (рис. 6.5).

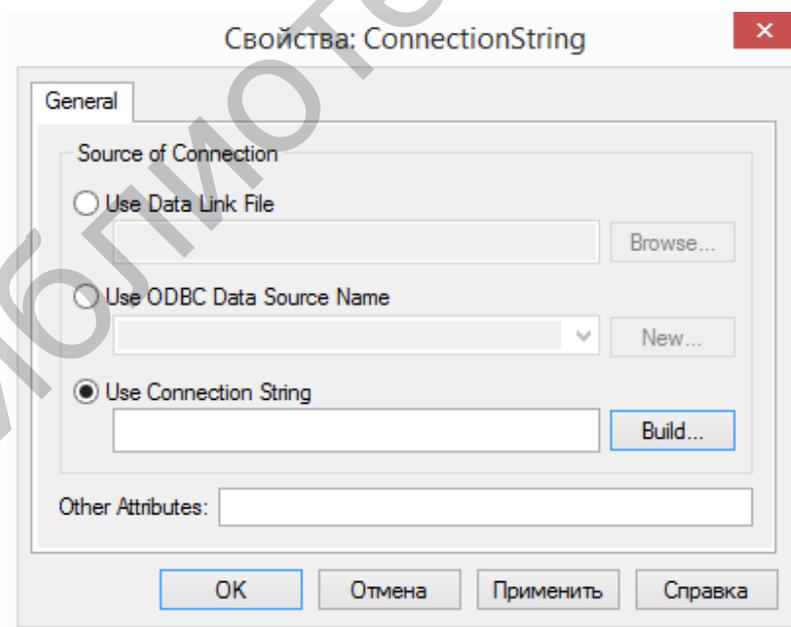


Рис. 6.5. Свойства ADO Data Control

Выберите поставщика данных Microsoft OLE DB Provider for ODBC Drivers (рис. 6.6) и нажмите *Далее*.

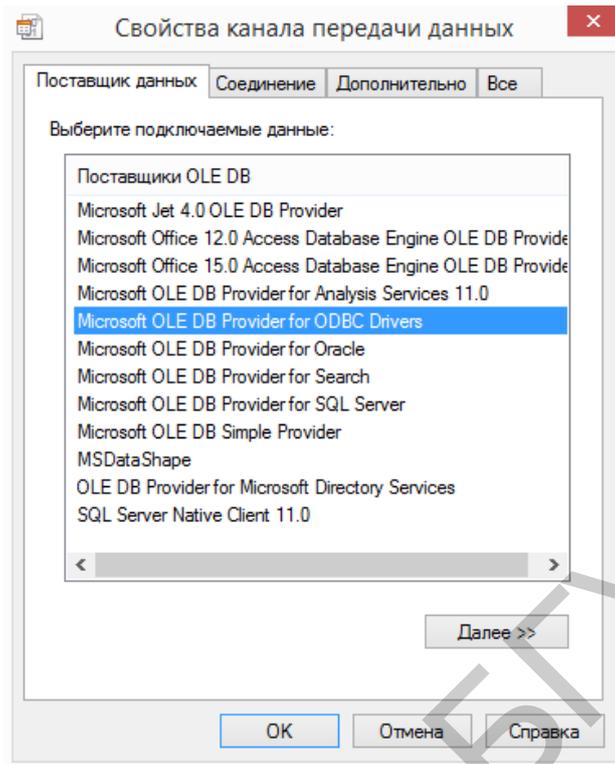


Рис. 6.6. Выбор поставщика данных

В появившемся окне **Выбор источника данных** (рис. 6.7) выберите источник данных `test_database_laba6` и нажмите ОК.

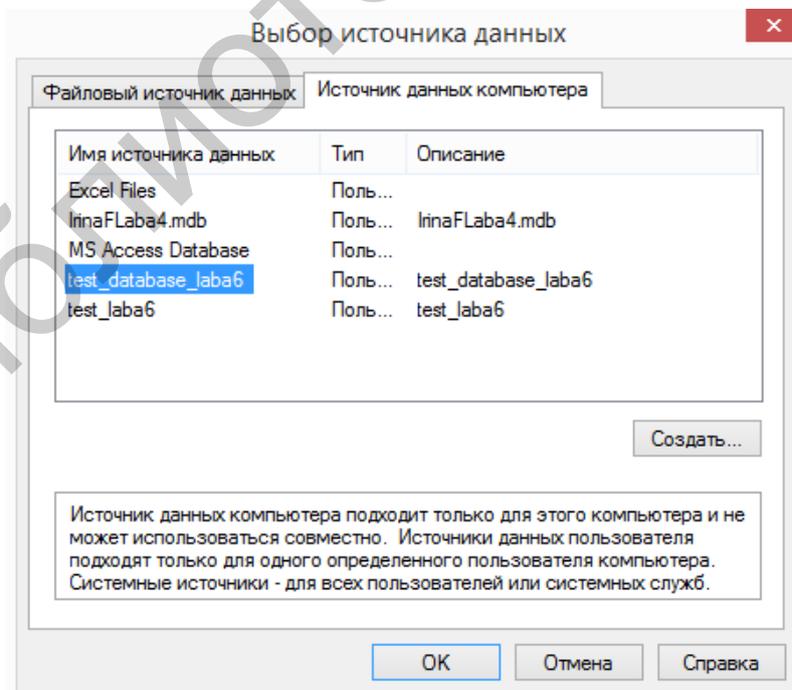


Рис. 6.7. Выбор источника данных

Зайдите в свойства ADO Data Control на вкладку RecordSource. Выберите значение Command Type: 1 – adCmdText. В поле Comand Text (SQL) введите строку SELECT \* FROM apteka (рис. 6.8).

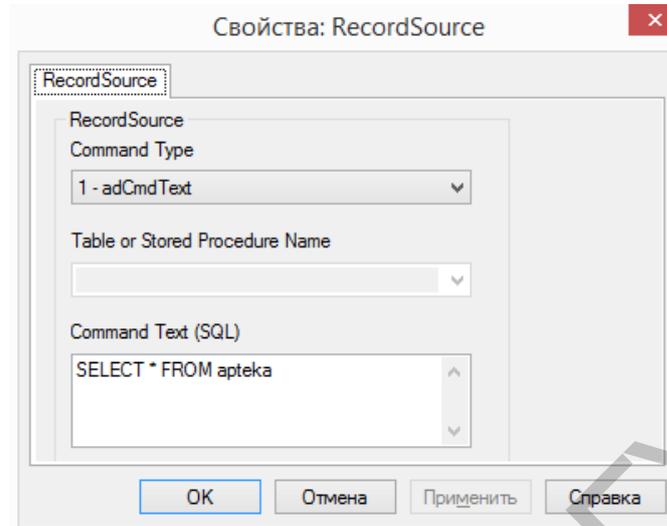


Рис. 6.8. Свойства ADO Data Control

Свяжите элементы управления DataGrid Control и ADO Data Control. В свойствах DataGrid Control на вкладке **Данные** поставьте пометки напротив следующих свойств: AllowAddNew, AllowDelete, AllowUpdate для добавления, удаления и редактирования записей (рис. 6.9).

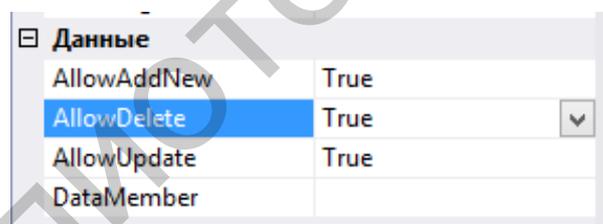


Рис. 6.9. Свойства DataGrid Control

В свойствах DataGrid Control на вкладке **Прочее** в параметре **Источник данных** выберите IDC\_ADODC1 (рис. 6.10).

Прочее	
(About)	Невозможно выполнить
(Имя)	IDC_DATAGRID1 (Active)
Видимый	True
Группа	False
ИД	IDC_DATAGRID1
Источник данных	IDC_ADODC1
Отключено	False
Поле данных	
Справка	True

Рис. 6.10. Свойства DataGrid Control

Организацию сортировки и фильтрации данных начните с объявления ассоциативных переменных. В *Мастере классов* выберите *Переменные-члены* и создайте переменные как показано на рис. 6.11.

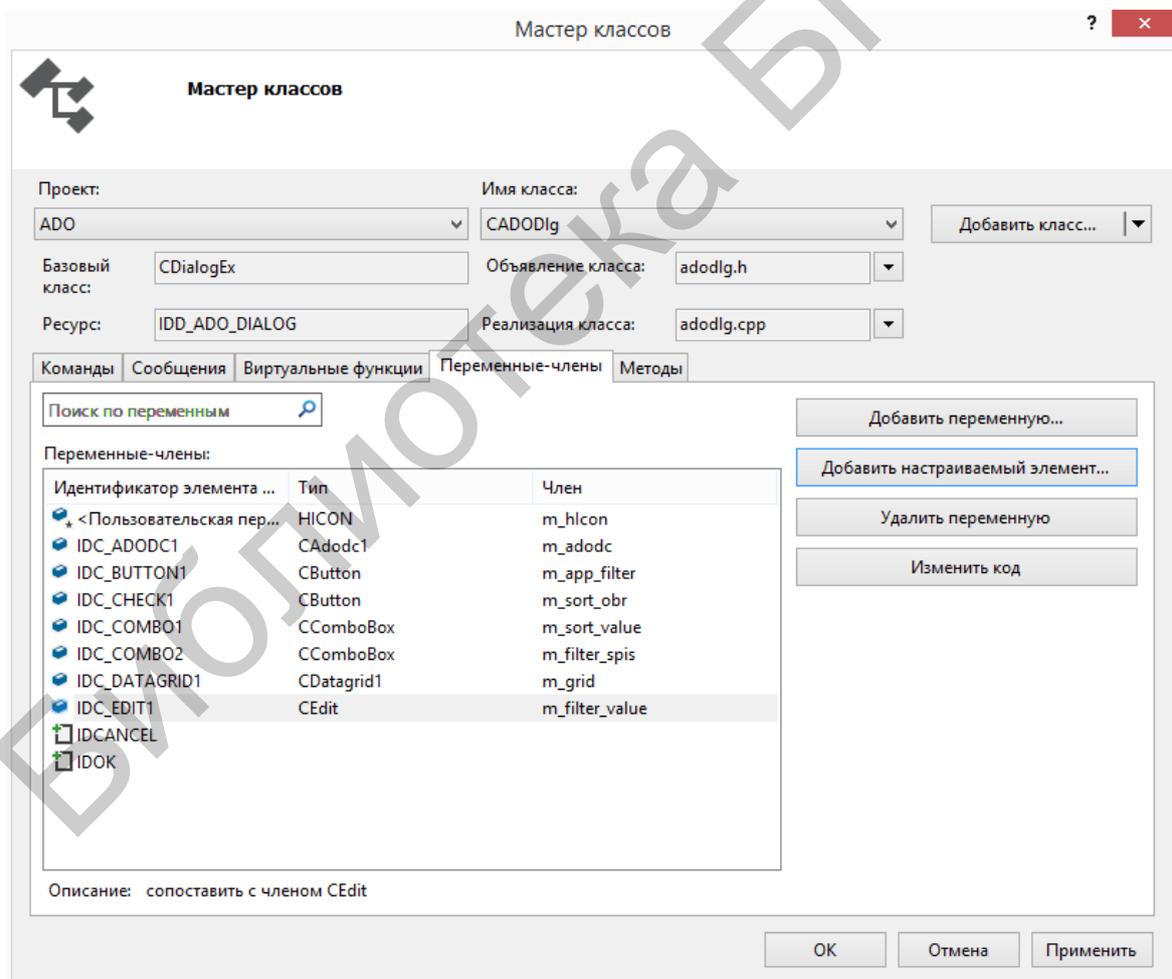


Рис. 6.11. Создание ассоциативных переменных

Инициализация данных в списках диалогового окна выполняется в методе OnInitDialog(), в котором необходимо заполнить списки сортировки и фильтрации, чтобы к моменту отображения диалогового окна они содержали информацию. Воспользуйтесь для этого методом AddString() объектов m\_sort\_value и m\_filter\_spis:

```

BOOL CADODlg::OnInitDialog()
{
    CDialogEx::OnInitDialog();
    ASSERT((IDM_ABOUTBOX & 0xFFF0) == IDM_ABOUTBOX);
    ASSERT(IDM_ABOUTBOX < 0xF000);
    CMenu* pSysMenu = GetSystemMenu(FALSE);
    if (pSysMenu != NULL)
    {
        BOOL bNameValid;
        CString strAboutMenu;
        bNameValid = strAboutMenu.LoadString(IDS_ABOUTBOX);
        ASSERT(bNameValid);
        if (!strAboutMenu.IsEmpty())
        {
            pSysMenu->AppendMenu(MF_SEPARATOR);
            pSysMenu->AppendMenu(MF_STRING, IDM_ABOUTBOX, strAboutMenu);
        }
    }
    SetIcon(m_hIcon, TRUE); // Крупный значок
    SetIcon(m_hIcon, FALSE); // Мелкий значок

    m_sort_value.AddString("unsort");
    m_sort_value.AddString("lekarstvo");
    m_sort_value.AddString("nazn");
    m_sort_value.AddString("recept");
    m_sort_value.AddString("kol");
    m_sort_value.AddString("cena");
    m_sort_value.AddString("proizv");

    m_filter_spis.AddString("unfilter");
    m_filter_spis.AddString("lekarstvo");
    m_filter_spis.AddString("nazn");
    m_filter_spis.AddString("recept");
    m_filter_spis.AddString("kol");
    m_filter_spis.AddString("cena");
    m_filter_spis.AddString("proizv");

    return TRUE;
}

```

Теперь для работы сортировки создайте в *Мастере обработчика события* метод OnSelchangeCombo1() для определения критерия сортировки (рис. 6.12).

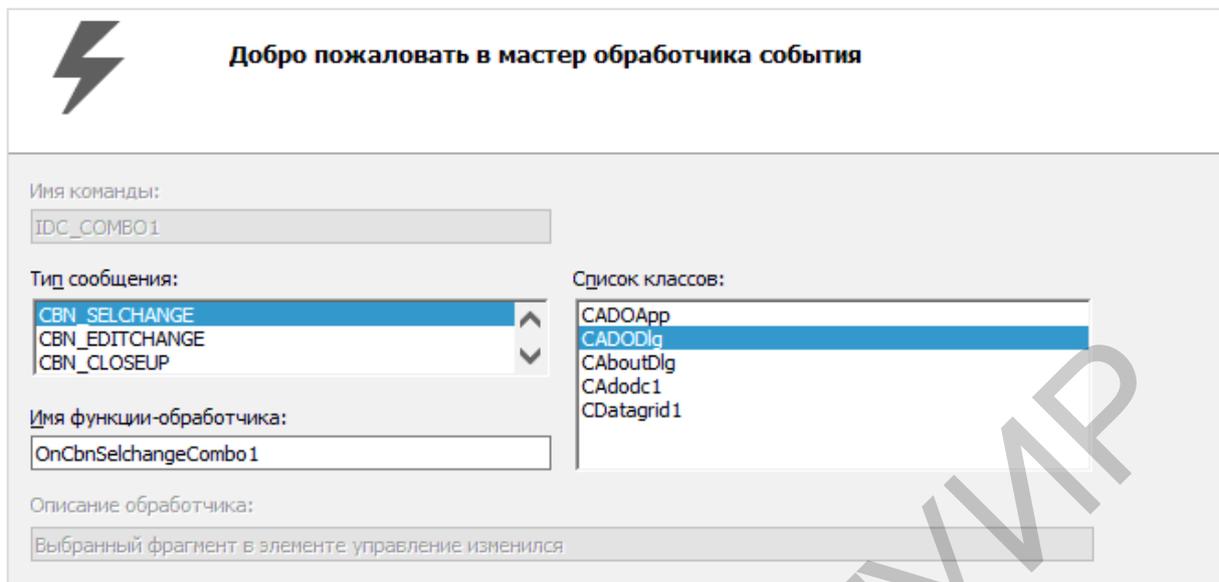


Рис. 6.12. Создание метода OnSelchangeCombo1()

Добавьте в метод OnSelchangeCombo1() следующее:

```
void CADODlg::OnCbnSelchangeCombo1 ()
{
    CString str1, str2;
    str1 = "SELECT * FROM apteka";
    if (m_sort_value.GetCurSel() == 0)
        str2 = "";
    else
    {
        m_sort_value.GetLBText(m_sort_value.GetCurSel(), str2);
        str2 = (CString) " ORDER BY " + str2;
        if (m_sort_obr.GetCheck())
            str2 += " DESC";
    }
    str1 += str2;
    m_adodc.put_RecordSource(str1);
    m_adodc.Refresh();
}
```

Выбор значения из ComboBox формирует условие отбора. Если необходимо сортировать в обратном порядке, то следует отметить Check Box, тем самым, добавляя дополнительное условие. По завершении формирования условия отбора необходимо передать получившуюся строку в DataControl=>RecordSource, после чего обновите базу данных.

Для работы фильтра создайте событие OnVnClickedButton1 в *Мастере обработчика события* (рис. 6.13).

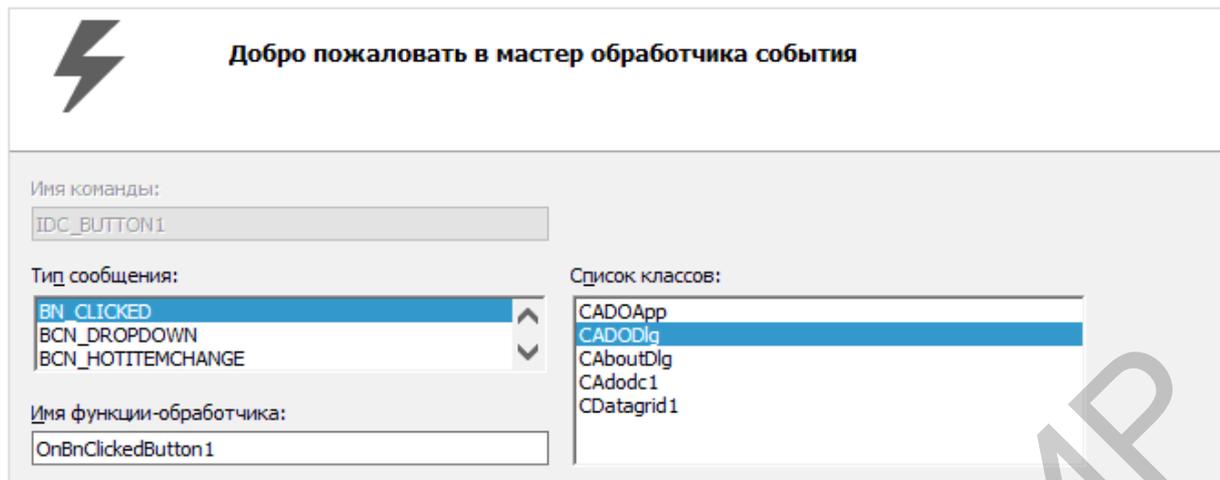


Рис. 6.13. Создание события OnBnClickedButton1

```

void CADODlg::OnBnClickedButton1()
{
    CString str1, str2, str3;
    str1 = "SELECT * FROM apteka ";
    m_filter_spis.GetLBText(m_filter_spis.GetCurSel(), str2);
    if (str2 == "unfilter")
    {
        str2 = "";
        str3 = "";
    }
    else
    if (str2 != "")
    {
        m_filter_value.GetWindowText(str3);

        str1 += "WHERE ";
        str1 += str2;
        str1 += " LIKE ";
        str1 += "!%";
        str1 += str3;
        str1 += "%'";
    }
    m_adodc.put_RecordSource(str1);
    m_adodc.Refresh();
}

```

Из ComboBox2 выбирается критерий фильтра, а значение вводится в поле Edit. По нажатии кнопки App\_filter формируется строка запроса, которая передается в DataControl=>RecordSource и база данных обновляется.

Результаты работы программы представлены на рис. 6.14.

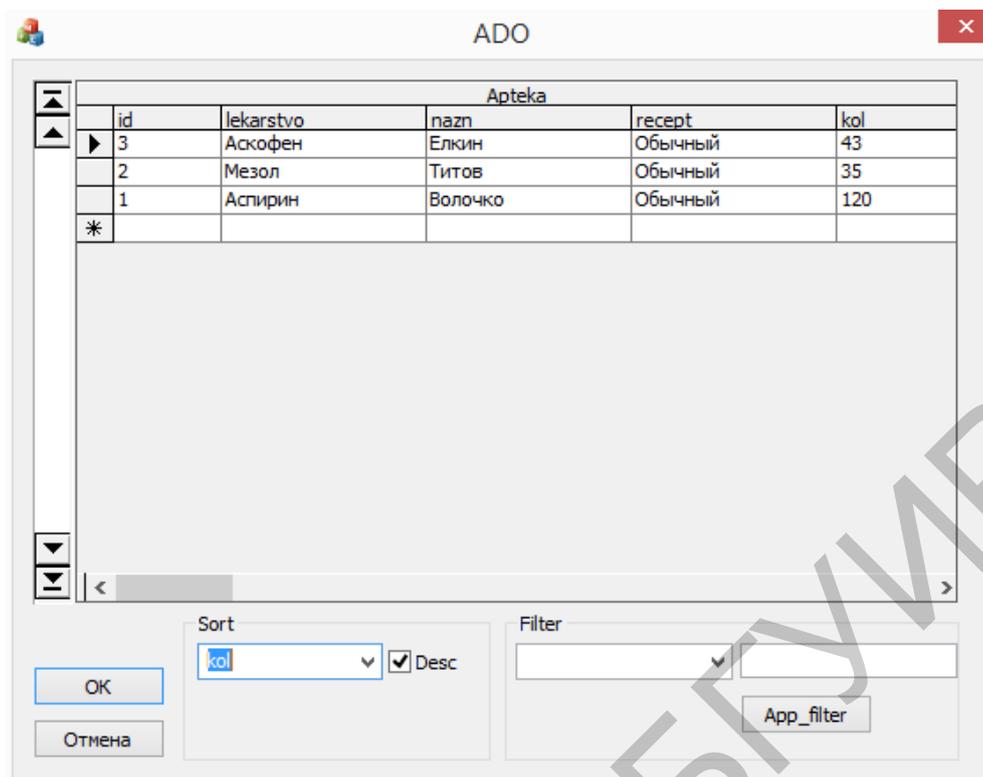


Рис. 6.14. Отсортированная в обратном порядке база данных

### Задание

Разработать приложение доступа к базе данных с использованием технологии ADO, для чего использовать источник данных из лабораторной работы №4.

## ЛАБОРАТОРНАЯ РАБОТА №7 ИСПОЛЬЗОВАНИЕ ПОТОКОВ В ПРИЛОЖЕНИИ

Цель работы – ознакомиться с принципами организации многозадачности в Win32-приложениях, изучить способы синхронизации работы потоков и доступа к данным.

### Методические указания

**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);
```

При успешном завершении функция `AfxBeginThread` возвращает указатель на объект потока, в противном случае возвращает нуль.

**Синхронизация потоков.** Иногда при работе с несколькими потоками или процессами появляется необходимость синхронизировать выполнение двух или более из них. Причина этого чаще всего заключается в том, что два или более потоков могут требовать доступ к разделяемому ресурсу, который реально не может быть предоставлен сразу нескольким потокам.

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.

Вторым типом объектов синхронизации является исключающий (mutex) семафор. Он предназначен для полного ограничения доступа к ресурсу, чтобы в любой момент времени к ресурсу мог обратиться только один процесс или поток.

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 выполнение определенных участков кода программы несколькими потоками одновременно. Для этого данные участки должны быть объявлены как критическая секция (critical section).

В MFC механизм синхронизации поддерживается с помощью следующих классов, порожденных от класса CSyncObject: CCriticalSection – критическая секция, CEvent – объект события, CMutex – исключающий семафор, CSemaphore – классический семафор.

Кроме этих классов в MFC определены два вспомогательных класса синхронизации: 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. Далее будем рассматривать только класс CSingleLock.

**Разработка приложения.** Рассмотрим пример создания двух потоков для однодокументного приложения Example при обработке сообщения о выборе пользователем пункта меню *Запуск потока* меню *Поток*. В качестве родительского потока выступает главный поток приложения. Поток 1 после запуска осуществляет 100-кратный вывод некоторой строки в окно приложения с задержкой 650 мс, поток 2 каждые две секунды 50 раз выдает звуковой сигнал и сообщение.

Для создания приложения Example выполните следующие действия.

Создайте новый проект с именем Example. Используя редактор ресурсов, добавьте в меню приложения новое меню *Поток*. Поместите в него команду с названием *Запуск потока* и идентификатором ID\_STARTTHREAD (рис. 7.1).

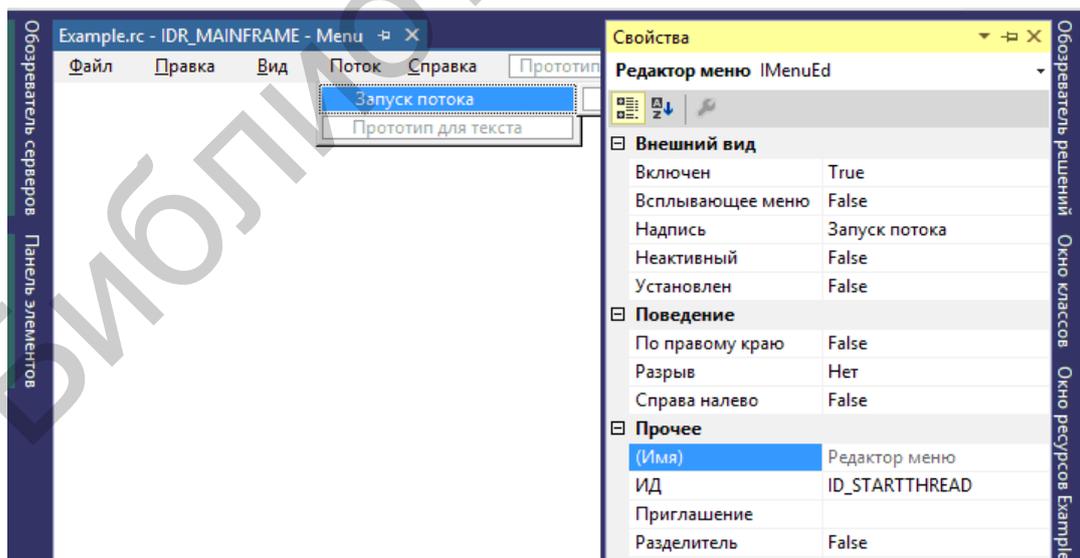


Рис. 7.1. Добавление нового пункта меню

Добавьте функцию OnStartthread() для обработки событий. С помощью *Мастера классов* свяжите команду ID\_STARTTHREAD с функцией обработки

сообщения OnStartthread(), для этого щелкните правой кнопкой мыши на **Запуск потока** и выберите пункт меню **Добавить обработчик событий....** Перед добавлением этой функции убедитесь, что в поле **Список классов** выбрано значение CExampleView.

Далее откройте функцию OnStartthread() и добавьте следующий код.

```
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.

По такому же принципу создайте в меню **Поток** пункт **Запуск потока 1** и добавьте обработчик события OnStartthread1, в который поместите код

```
AfxBeginThread(MyThread3, this);
```

Окончательный фрагмент кода в файле ExampleView.cpp представлен ниже.

```
UINT MyThread1(LPVOID pParam); // объявление функции потока 1  
UINT MyThread2(LPVOID pParam); // объявление функции потока 2  
UINT MyThread3(LPVOID pParam);  
void CExampleView::OnStartthread() // обработка 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  
void CExampleView::OnStartthread1()  
{  
AfxBeginThread(MyThread3, this);  
}  
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 мс
CRect r;
ptrView->GetClientRect(&r); // получить клиентскую область
dc->SetTextColor(RGB(255,0,0)); // задание цвета окна
dc->TextOut(rand()%r.Width()+rand()%100,rand()%r.Height(),
           CString("R"),1);
}
return 0;
}
// определение функции потока 2
UINT MyThread2(LPVOID pParam)
{
for(int i=0; i<50; i++)
{
Sleep(2000); // Задержка на 2000 мс
AfxMessageBox(CString("MyThread")); // Вывод сообщения
MessageBeep(0); // Подача звукового сигнала
}
return 0;
}
UINT MyThread3(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 мс
CRect r;
ptrView->GetClientRect(&r); // получить клиентскую область окна
dc->SetTextColor(RGB(0,255,0));
dc->TextOut(rand()%r.Width()+rand()%100,rand()%r.Height(),
           CString("G"),1); // вывод текста
}
return 0;
}

```

Откомпилируйте и запустите приложение. Не забудьте при компиляции установить в **Проект=>Свойства** опцию многопоточкового приложения, как это показано на рис. 7.2.

Иногда бывает необходимо приостановить поток на заданное количество миллисекунд. Это можно сделать, вызвав API-функцию Sleep. Поток выполняется до завершения своей потоковой функции. Поток может также «завершить сам себя» с помощью функции AfxEndThread. Параметр этого метода содержит статус завершения потока. Как правило, лучше давать потоку возможность нормально завершиться одновременно с потоковой функцией.

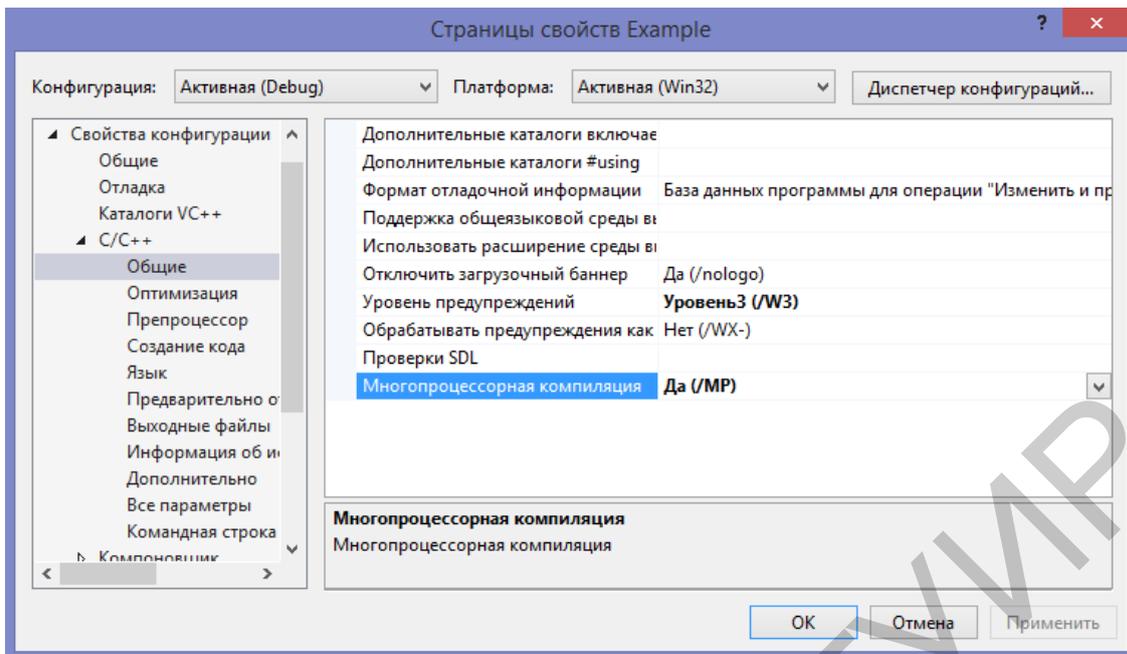


Рис. 7.2. Установка опции многопоточкового приложения

В результате процесс работы приложения должен иметь вид, как на рис. 7.3.

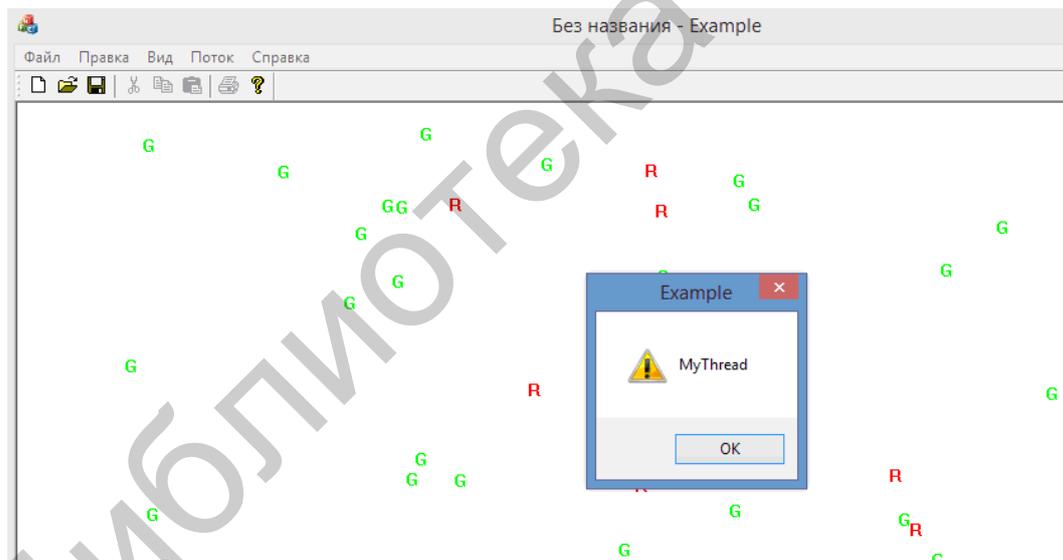


Рис. 7.3. Пример работы приложения

Рассмотрим процесс создания и использования объектов синхронизации.

**Работа с семафорами.** Модифицируйте приложение Example, добавив в него функции, использующие семафор. Для этого добавьте в меню **Поток** пункт **Семафор**. 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.

Доступ к разделяемому ресурсу осуществите 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 `SomeResource.h`:

```
#include "afxmt.h"
class CSomeResource{
private:
    CSemaphore* semaphore;
public:
    CSomeResource();
    ~CSomeResource();
    void UseResource();
};
```

Файл реализации класса `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 пять секунд и вновь его освобождает.

Модифицируйте приложение `Example` следующим образом. 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(CString("Thread1 had access."));
    return 0;
}
UINT ThreadProc2(LPVOID pParam)
{
    someResource.UseResource();
    AfxMessageBox(CString("Thread2 had access."));
    return 0;
}
UINT ThreadProc3(LPVOID pParam)
{
    someResource.UseResource();
    AfxMessageBox(CString("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 **Поток=>Семафор**. Приблизительно через пять секунд появятся два окна сообщений, информирующие о том, что первый и второй потоки получили доступ к защищенному ресурсу. Еще через пять секунд появится третье окно сообщений, в котором говорится о том, что третий поток также получил доступ к ресурсу.

**Работа с объектом события.** Объект события используется для оповещения процесса или потока о том, что произошло некоторое событие. Для работы с такими объектами предназначен класс CEvent.

Когда объект события создан, то поток, ожидающий данное событие, должен создать объект класса CSingleLock, для которого затем следует вызвать метод Lock. При этом выполнение данного потока останавливается до тех пор, пока не произойдет ожидаемое событие. Для сигнализации о том, что событие произошло, предназначена функция SetEvent класса CEvent. Она переводит

объект события в состояние «сигнализирует». При этом поток, ожидающий событие, выйдет из остановленного состояния (вызванный им метод Lock завершится) и поток продолжит свое выполнение.

Чтобы продемонстрировать работу с объектами события, дополним наше приложение следующими функциями. Создадим два пункта меню: **Запуск потока 2** и **Остановка потока 2**. По пункту **Запуск потока 2** должен запускаться процесс MyThread2, по пункту **Остановка потока 2** должен заканчиваться процесс MyThread2 с выдачей сообщения «MyThread2 ended». Если запуск процесса легко осуществить с помощью объекта события, то завершение процесса легче реализовать с помощью глобальной переменной.

Выполните следующие действия для реализации объекта события и использования глобальной переменной. С помощью редактора ресурсов добавьте пункт меню Thread2 и в него команды **Запуск потока 2** и **Остановка потока 2**. Присвойте этим командам идентификаторы ID\_STARTTHREAD2 и ID\_ENDTHREAD2.

Теперь щелкните правой кнопкой мыши на **Запуск потока 2** и выберите **Добавить обработчик событий...**, свяжите команду ID\_STARTTHREAD2 с функцией обработки сообщения OnStartthread2() и таким же образом команду ID\_ENDTHREAD2 с функцией обработки сообщения OnEndhread2().

Добавьте в начало файла ExampleView.cpp после строки

```
#include "SomeResource.h"
```

строку подключения заголовочного файла для работы с классами объектов синхронизации

```
#include "afxmt.h"
```

Включите в начало файла ExampleView.cpp сразу же после объявления

```
CSomeResource someResource
```

следующие строки:

```
volatile bool keeprunning;  
CEvent threadStart;  
CEvent threadEnd;
```

Добавьте в функцию UINT MyThread2(LPVOID pParam) в файле ExampleView.cpp перед циклом for строки

```
CSingleLock syncObjStart(&threadStart);  
syncObjStart.Lock();
```

Первая функция создает объект `syncObjStart` класса `CSingleLock` для объекта события `threadStart`. Вторая функция вызывает метод `Lock()` для этого объекта. Выполнение данного потока приостанавливается до тех пор, пока не произойдет событие для этого объекта.

Добавьте в функцию `UINT MyThread2(LPVOID pParam)` в файле `ExampleView.cpp` в тело цикла после строки

```
MessageBeep(0);
```

строки

```
if (keepunning == FALSE)
{
    AfxMessageBox(CString("MyThread2 ended"));
    break;
}
```

Теперь при каждом проходе цикла будет осуществляться проверка глобальной переменной и, если она станет равна `FALSE`, цикл прервется и поток завершит свою работу.

Добавьте в функцию `void CExampleView::OnStartthread2()` строки

```
keepunning = TRUE;
threadStart.SetEvent();
```

Первая строка устанавливает начальное значение переменной `keepunning`. Вторая функция устанавливает объект события в состояние «сигнализирует» (событие произошло). После ее выполнения метод `Lock()`, который ждет этого события, завершает свою работу и выполняются следующие функции.

Добавьте в функцию `void CExampleView::OnEndthread2()` строку

```
keepunning = FALSE;
```

Теперь очередной цикл в функции `MyThread2` прервется и поток завершит свою работу.

**Использование критической секции.** Критическая секция (`Critical Section`) – это участок кода, в котором поток получает доступ к ресурсу (например, переменной), который доступен из других потоков.

Для создания в программе, использующей библиотеку MFC, объекта критической секции необходимо создать экземпляр объекта класса `CCritical Section`. Когда в программе необходимо получить доступ к данным, защищенным критической секцией, вызывается метод `Lock()` объекта этой критической секции. Если объект критической секции 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()` для освобождения объекта критической секции.

Для ознакомления с объектом критической секции создайте в разработанном приложении `Example` новый пункт `Critical Section` в меню **Поток**. При выборе этого пункта будут запускаться две потоковые функции: записи элементов в массив и считывание элементов из массива. Операции чтения и записи в массив защищены критическими секциями.

С помощью редактора ресурсов добавьте новый пункт `Critical Section` в меню **Поток**. Присвойте этому пункту идентификатор `ID_CRITICALSECTION`. С помощью свойств свяжите команду `ID_CRITICALSECTION` с функцией обработки сообщения `void CExampleView::OnCriticalsection()`.

Введите приведенные ниже операторы в функцию `OnCriticalsection()`:

```
AfxBeginThread(WriteThreadProc, this);  
AfxBeginThread(ReadThreadProc, this);
```

Добавьте в проект два новых пустых файла `CountArray.h` и `CountArray.cpp`. Добавьте в файл `CountArray.h` следующие строки:

```
#include "afxmt.h"  
class CCountArray  
{  
private:  
    int array[10];  
    CCriticalSection criticalSection;  
public:  
    CCountArray() {};  
    ~CCountArray() {};  
    void SetArray(int value);  
    void GetArray(int dstArray[10]);  
};
```

В начале файла к программе подключается файл заголовка библиотеки MFC `afxmt.h`, обеспечивающий доступ к классу `CCriticalSection`. В объявлении класса `CCountArray` выполняется объявление целочисленного массива из десяти элементов, предназначенного для хранения защищаемых критической секцией данных, а также объявляется объект критической секции `criticalSection`. Открытые методы класса `CCountArray` включают конструктор и деструктор, а также две функции для чтения и записи массива. Добавьте в пустой файл `CountArray.cpp` следующий текст:

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

```

void CCountArray::SetArray(int value)
{
    criticalSection.Lock();
    for (int x=0; x<10; ++x)
        array[x] = value;
    criticalSection.Unlock();
}
void CCountArray::GetArray(int dstArray[10])
{
    criticalSection.Lock();
    for (int x=0; x<10; ++x)
        dstArray[x] = array[x];
    criticalSection.Unlock();
}

```

Каждый метод класса CCountArray обеспечивает захват и освобождение объекта критической секции. Это означает, что любой поток может вызвать эти методы, абсолютно не заботясь о синхронизации потоков. Например, если поток 1 вызовет функцию SetArray(), первое, что сделает эта функция, – будет вызов criticalSection.Lock(), которая передаст объект критической секции во владение этому потоку. Затем весь цикл for выполняется в полной уверенности, что его работа не будет прервана другим потоком. Если в это время поток 2 вызовет функцию SetArray() или GetArray(), то очередной вызов criticalSection.Lock() приостановит работу потока до тех пор, пока поток 1 не освободит объект критической секции. А это произойдет тогда, когда функция SetArray() закончит выполнение цикла for и вызовет criticalSection.Unlock(). Затем система возобновит работу потока 2, передав ему во владение объект критической секции.

Откройте файл CExampleView.cpp и добавьте в него после строки

```
#include "afxmt.h"
```

строку

```
#include "CountArray.h"
```

Добавьте в начало этого же файла после строки

```
volatile bool keeprunning;
```

строку

```
CCountArray countArray;
```

Добавьте в файл ExampleView.cpp перед функцией

```
void CExampleView::OnCriticalSection()
```

следующие функции:

```
UINT WriteThreadProc(LPVOID param)
{
for(int x=0; x<10; ++x)
  { countArray.SetArray(x);
    Sleep(1000);
  }
return 0;
}
UINT ReadThreadProc(LPVOID param)
{int array[10];
for (int x=0; x<20; ++x)
  {countArray.GetArray(array);
  char str[50];
  str[0] = 0;
  for (int i=0; i<10; ++i)
    {
      int len = strlen(str);
      sprintf(&str[len], "%d ", array[i]);
    }
  AfxMessageBox(CString(str));
}
return 0;
}
```

Откомпилируйте новую версию приложения Example и запустите ее на выполнение. На экране раскроется главное окно приложения. Для запуска процесса выберите команду **Помок=>Critical section**. Первым появится окно сообщений, отображающее текущие значения элементов защищенного массива. Каждый раз при закрытии оно будет появляться вновь, отображая обновленное содержимое массива. Всего вывод окна будет повторяться 20 раз. Значения, отображаемые в окне сообщений, будут зависеть от того, насколько быстро вы будете его закрывать. Поток 1 записывает новые значения в массив каждую секунду, причем даже тогда, когда вы просматриваете содержимое массива с помощью потока 2.

Обратите внимание на одну важную деталь: поток 2 ни разу не прервал работу потока 1 во время изменения им значений в массиве. На это указывает идентичность всех десяти значений элементов массива. Если бы работа потока 1 прерывалась во время модификации массива, то десять значений массива были бы неодинаковы.

Поток 1 с именем WriteThreadProc() вызывает функцию-член SetArray() класса CCountArray десять раз за один цикл for. В каждом цикле функция SetArray() захватывает объект критической секции, заменяет содержимое массива переданным ей числом и вновь освобождает объект критической

секции. Поток 2 `ReadThreadProc()` также пытается захватить объект критической секции, чтобы иметь возможность сформировать строку на экране, содержащую текущие значения элементов массива. Но, если в данный момент поток `WriteThreadProc()` заполняет массив новыми значениями, поток `ReadThreadProc()` вынужден будет ждать.

### Задания

1. Разработать приложение, которое может запускать один (поток Red), два (потoki Red и Green) или три потока (потoki Red, Green, Blue). Каждый из потоков рисует прямоугольники своим цветом и в своей области окна представления. Предусмотреть команду остановки выполнения потоков.

2. Разработать приложение, которое может запускать один (поток Red), два (потoki Red и Green) или три потока (потoki Red, Green, Blue). Каждый из потоков выводит символ своим цветом и в своей области окна представления. Предусмотреть команду остановки выполнения потоков.

3. Разработать приложение, которое может запускать потоки Red, Green, и Blue. Каждый из потоков рисует окружности своим цветом. Предусмотреть команду остановки выполнения потоков.

4. Организовать доступ к файлу на диске из двух различных потоков. В файле хранится информация о банковском счете. Поток 1 увеличивает значение счета на единицу в одну секунду. Поток 2 выводит в окно `AfxMessageBox` величину счета в произвольный момент времени. Для синхронизации доступа к данным использовать объект критической секции. Предусмотреть команду остановки выполнения потоков.

5. Организовать вывод в главное окно приложения фразу, состоящую из двух частей. Первая часть фразы формируется потоком 1, вторая часть – потоком 2. Для синхронизации доступа к данным использовать объект события. Предусмотреть команду остановки выполнения потоков.

6. Организовать запись в файл строки, состоящей из двух частей. Первая часть строки есть «Сумма =». Вторая часть строки есть число, которое формируется потоком 1 путем добавления единицы каждую секунду. Поток 2 выводит полученную строку в файл. Для синхронизации доступа к данным использовать объект семафора. Предусмотреть команду остановки выполнения потоков.

7. Разработать приложение, которое выводит диалоговую панель с кнопкой Start и списком List box. При нажатии кнопки Start организуется запуск потока, который заполняет список некоторыми значениями.

8. Разработать приложение, которое выводит диалоговую панель с кнопкой Array и списком List box. При нажатии кнопки Array организуется запуск четырех потоков. Первый запускает функцию обнуления массива. Второй выводит обнуленный массив в List box. Третий заполняет массив некоторыми значениями. Четвертый выводит заполненный массив в List box. Синхронизацию потоков осуществить с помощью семафора.

## ЛАБОРАТОРНАЯ РАБОТА №8 РАЗРАБОТКА СЕТЕВОГО ПРИЛОЖЕНИЯ С ИСПОЛЬЗОВАНИЕМ WINDOWS SOCKETS

Цель работы – научиться использовать возможности класса `CAsyncSocket` для организации сетевого приложения.

### Методические указания

Основным объектом, используемым в большинстве приложений для работы с сетью, является сокет. Сокет представляет собой объект, позволяющий осуществлять отправку и получение сообщений, которые будут пересылаться от одного компьютера к другому. Для того чтобы открыть сокет, необходимо знать, где расположен компьютер, на котором работает приложение, и номер порта, на котором это приложение ожидает вызов.

Приложения могут использовать множество сетевых возможностей, и все эти возможности используют свойства интерфейса `Winsock`. В данной лабораторной работе рассмотрим возможности класса `CAsyncSocket` для организации сетевого приложения.

Класс `CAsyncSocket` инкапсулирует асинхронные вызовы `Winsock`. Он содержит набор функций, использующих `Winsock API` (табл. 8.1).

Таблица 8.1

Методы класса `CAsyncSocket`

Метод	Назначение
1	2
<code>Accept</code>	Обрабатывает запрос на соединение, который поступает на принимающий сокет, заполняя его информацией об адресе
<code>AsyncSelect</code>	Организует посылку сообщения <code>Windows</code> при переходе сокета в состояние готовности
<code>Attach</code>	Связывает дескриптор сокета с экземпляром класса <code>CAsyncSocket</code> , чтобы иметь возможность сформировать соединение с другим компьютером
<code>Bind</code>	Ассоциирует адрес с сокетом
<code>Close</code>	Закрывает сокет
<code>Connect</code>	Подключает сокет к удаленному адресу и порту
<code>Create</code>	Завершает процесс инициализации, начатый конструктором
<code>GetLastError</code>	Возвращает код ошибки сокета
<code>GetPeerName</code>	Определяет адрес IP и номер порта удаленного компьютера
<code>GetSockName</code>	Возвращает адрес IP и номер порта объекта <code>this</code>
<code>Listen</code>	Заставляет сокет следить за запросами на соединение
<code>OnAccept</code>	Обрабатывает сообщение <code>Windows</code> , которое формируется при приеме гнездом запроса на соединение. Часто переопределяется в производных классах
<code>OnClose</code>	Обрабатывает сообщение <code>Windows</code> , которое формируется при закрытии сокета. Часто переопределяется в производных классах
<code>OnConnect</code>	Обрабатывает сообщение <code>Windows</code> , которое формируется после установки соединения или после неудачной попытки соединиться

1	2
OnReceive	Обрабатывает сообщение Windows, которое формируется при появлении данных, которые можно прочесть с помощью Receive()
OnSend	Обрабатывает сообщение Windows, которое формируется при готовности гнезда принять данные, посылаемые с помощью Send()
Receive	Считывает данные с удаленного компьютера, к которому подключен сокет
Send	Посылает данные удаленному компьютеру
SetSockOpt	Устанавливает параметры сокета
ShutDown	Оставляет сокет открытым, но предотвращает дальнейшие вызовы Send() или Receive()

В качестве примера сетевого приложения создайте диалоговое приложение, которое сможет работать либо в качестве сервера, либо в качестве клиента. Это позволит проверить созданное приложение, если запустить две копии приложения по одной на каждом конце соединения. Эти две копии или могут быть расположены на одном компьютере, или же одна из копий может быть установлена на другом компьютере, тогда два приложения будут работать на различных компьютерах, передавая сообщения по сети.

После того как между приложениями будет установлено соединение, можно передавать сообщения от одного приложения другому. После того как сообщение послано, оно будет добавлено в список переданных сообщений. В свою очередь, каждое полученное сообщение будет помещено в список всех полученных сообщений.

Создайте новый MFC-проект Sock. В настройке дополнительных параметров приложения установите флажок *Сокеты Windows*, как показано на рис. 8.1.

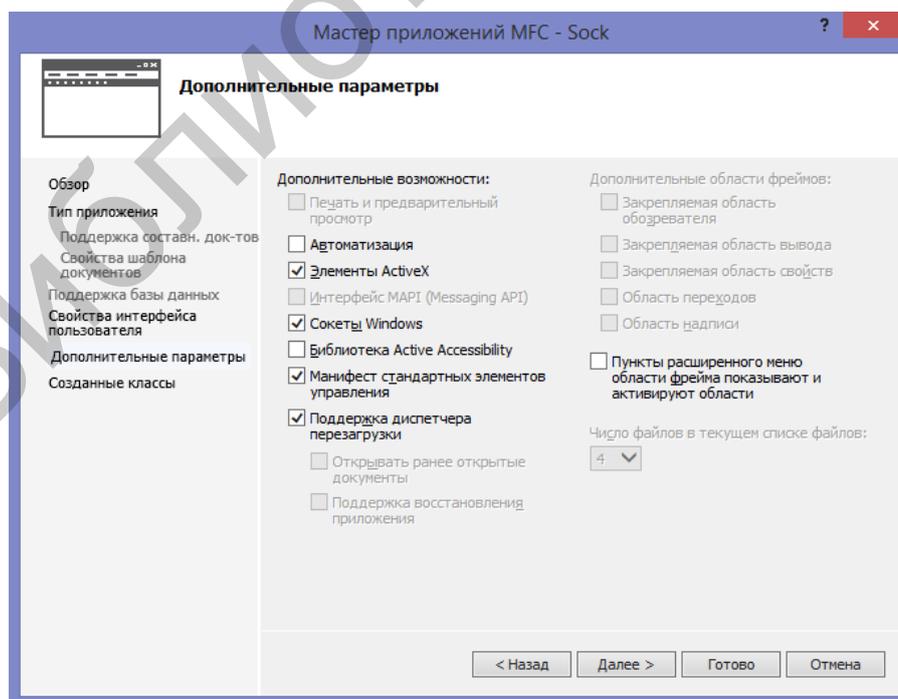


Рис. 8.1. Установка флажка *Сокеты Windows*

**Проектирование внешнего вида приложения.** После того как создан каркас приложения, приступите к созданию внешнего вида окна. Здесь нужно создать набор радиокнопок, с помощью которых можно установить параметры приложения, а именно, является ли данная копия приложения клиентской или серверной. Затем понадобится пара окон для редактирования текста, в которых нам следует указать имя компьютера и номер порта, на котором сервер будет прослушивать. Создайте командную кнопку, которая будет заставлять сервер приступать к прослушиванию на сокете или же заставлять клиента устанавливать соединение с сервером, а также кнопку, которая позволит закрывать соединение. Кроме того, необходимо определить окно для редактирования текста, в которое будет вводиться сообщение, предназначенное для передачи другому приложению, и кнопку, которая будет осуществлять такую передачу. Наконец, нужно иметь пару окон для списка, в которые будут помещаться переданные и полученные сообщения.

Добавьте элементы и измените их свойства:

1. Откройте вкладку с ресурсами приложения (Ctrl + Shift + E).
2. Откройте вкладку Sock.rc, далее вкладку Dialog и дважды щелкните левой кнопкой мыши на элементе IDD SOCK\_DIALOG, который соответствует диалоговому окну приложения.

3. В результате откроется диалоговое окно. Справа найдите вкладку **Панель элементов** (если не можете найти вкладку, откройте ее с помощью Ctrl + Alt + X). Добавьте элементы управления в соответствии с тем, как показано на рис. 8.2.

4. В соответствии с табл. 8.2 задайте свойства элементов управления, после чего в соответствии с табл. 8.3 создайте переменные для элементов управления; для всех элементов управления оставьте область видимости public (для создания переменных откройте пункт меню **Проект=>Добавить переменную...**).

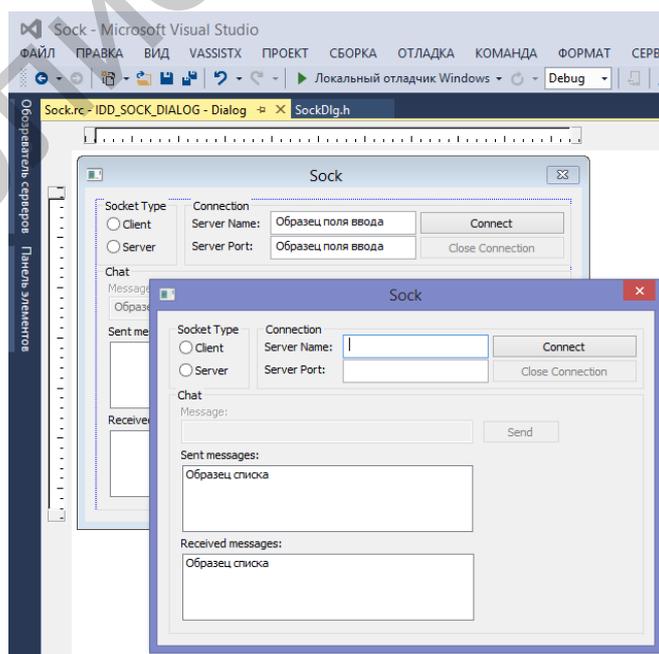


Рис. 8.2. Расположение элементов управления

Параметры элементов управления

Объект	Свойства	Значение
Group Box	ID	IDC_STATICTYPE
	Caption	Socket Type
RadioButton	ID	IDC_RCLIENT
	Caption	&Client
	Group	True
RadioButton	ID	IDC_RSERVER
	Caption	&Server
Static Text	ID	IDC_STATICNAME
	Caption	Server &Name:
Edit Control	ID	IDC_ESERVNAME
Static Text	ID	IDC_STATICPORT
	Caption	Server &Port:
Edit Control	ID	IDC_ESERVPORT
Button	ID	IDC_BCONNECT
	Caption	C&onnect
Button	ID	IDC_BCLOSE
	Caption	C&lose Connection
	Disabled	True
Static Text	ID	IDC_STATICMSG
	Caption	&Message:
	Disabled	True
Edit Control	ID	IDC_EMMSG
	Disabled	True
Button	ID	IDC_BSEND
	Caption	S&end
	Disabled	True
Static Text	ID	IDC_STATIC
	Caption	Sent messages:
List Box	ID	IDC_LSENT
	Tab Stop	False
	Selection	None
Static Text	ID	IDC_STATIC
	Caption	Received messages:
List Box	ID	IDC_LRECVD
	Tab Stop	False
	Sort	False
	Selection	None

После того как основа диалогового окна сконструирована, откройте *Мастер классов* и создайте переменные для элементов контроля в соответствии с переменными элементов управления (табл. 8.3).

Переменные элементов управления

Объект	Имя	Категория	Тип
IDC_BCONNECT	m_ctlConnect	Control	CButton
IDC_EMSG	m_strMessage	Value	CString
IDC_ESERVNAME	m_strName	Value	CString
IDC_ESERVPORT	m_iPort	Value	int
IDC_LRECVD	m_ctlRecvd	Control	CListBox
IDC_LSENT	m_ctlSent	Control	CListBox
IDC_RCLIENT	m_iType	Value	int

Чтобы иметь возможность использовать кнопку CONNECT повторно и поставить приложение-сервер «прослушивать» в ожидании соединения, нужно вставить функцию к радиокнопкам; текст, изображаемый на командной кнопке, зависит от того, какая выбрана радиокнопка. Чтобы вставить требуемую функцию, соответствующую сообщению о событии BN\_CLICKED для идентификатора IDC\_RCLIENT, используйте имя функции OnRType. Вставьте такую же функцию для события BN\_CLICKED для элемента управления с идентификатором IDC\_RSERVER. Функции имеют вид:

```
void CSockDlg::OnRType()
{
// Синхронизировать элементы управления в соответствии с
// переменными
UpdateData(TRUE);
// В каком мы режиме?
if (m_iType == 0) // Установить текст на кнопке
    m_ctlConnect.SetWindowText(TEXT("C&onnect"));
else
    m_ctlConnect.SetWindowText(TEXT("&Listen"));
}
void CSockDlg::OnRTypeserver()
{
// Синхронизировать элементы управления в соответствии
// с переменными
UpdateData(TRUE);
// В каком режиме приложение?
if (m_iType == 0) //Установить текст на кнопке
    m_ctlConnect.SetWindowText(TEXT("C&onnect"));
else
    m_ctlConnect.SetWindowText(TEXT("&Listen"));
}
```

Если сейчас скомпилировать и запустить приложение, можно будет выбирать режим работы приложения с помощью двух кнопок, а текст, появляющийся на командной кнопке, будет меняться в зависимости от того, какой установлен режим.

**Функции класса CAsyncSocket.** Чтобы иметь возможность перехватывать и отвечать на события сокета, необходимо создать свой собственный класс на основе класса CAsyncSocket. В этом классе будет содержаться его собственная версия функций для обработки событий, а также средства отражения событий на уровне класса диалогового окна. Здесь будет использоваться указатель на диалоговое окно родительского класса нашего класса сокета. Этот указатель будет использоваться для осуществления вызова функций каждого события сокета. Предварительно осуществите проверку наличия ошибок. Чтобы создать этот класс, откройте пункт меню **Проект=>Добавить класс...** В появившемся диалоговом окне необходимо выбрать категорию **MFC=>MFC-класс** и нажать кнопку **Добавить**. В диалоговом меню мастера создания нового класса задайте имя нового класса **CMySocket**, а в качестве базового класса выберите **CAsyncSocket** (рис. 8.3). Нажмите **Готово**. Новый класс вставлен в проект.

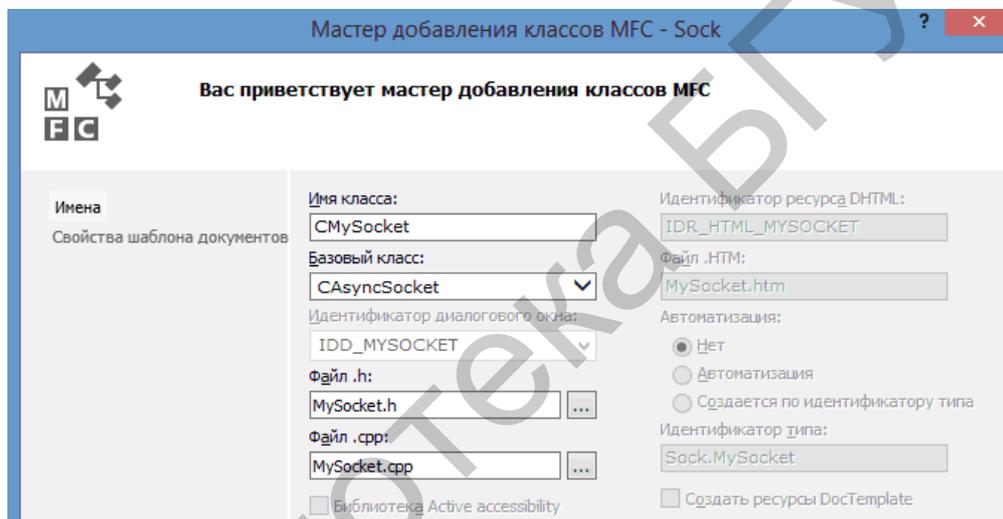


Рис. 8.3. Мастер добавления классов

После того как новый класс сокета создан, вставьте переменную в класс, который будет использоваться в качестве указателя на родительское диалоговое окно. Укажите тип переменной **CDialog\***, имя переменной **m\_pWnd**, доступ **private**. В классе необходимо определить метод, а значит, вставьте новую функцию в этот класс. Тип функции **void**, объявите функцию в виде **SetParent(CDialog\* pWnd)**, доступ **public**. Отредактируйте созданную функцию.

```
void CMySocket::SetParent(CDialog *pWnd)
{
    // устанавливаем указатель
    m_pWnd = pWnd;
}
```

В классе сокета создайте функции обработки событий. Для создания функции, соответствующей событию **OnAssert**, вставьте новую функцию в

класс сокета, тип функции void, опишите функцию в виде OnAccept(int nErrorCode), доступ protected. Отредактируйте код.

```
void CMySocket::OnAccept(int nErrorCode)
{
    if (nErrorCode == 0)
        // Нет, вызываем функцию OnAccept()
        ((CSockDlg*)m_pWnd)->OnAccept();
    CAsyncSocket::OnAccept(nErrorCode);
}
```

Вставьте подобные функции для событий OnConnect, OnClose, OnReceive и OnSend.

```
void CMySocket::OnClose(int nErrorCode)
{
    if (nErrorCode == 0)
        // Нет, вызываем функцию OnAccept()
        ((CSockDlg*)m_pWnd)->OnClose();
    CAsyncSocket::OnClose(nErrorCode);
}
void CMySocket::OnConnect(int nErrorCode)
{
    if (nErrorCode == 0)
        // Нет, вызываем функцию OnAccept()
        ((CSockDlg*)m_pWnd)->OnConnect();
    CAsyncSocket::OnConnect(nErrorCode);
}
void CMySocket::OnReceive(int nErrorCode)
{
    if (nErrorCode == 0)
        // Нет, вызываем функцию OnAccept()
        ((CSockDlg*)m_pWnd)->OnReceive();
    CAsyncSocket::OnReceive(nErrorCode);
}
void CMySocket::OnSend(int nErrorCode)
{
    if (nErrorCode == 0)
        // Нет, вызываем функцию OnAccept()
        ((CSockDlg*)m_pWnd)->OnBsend();
    CAsyncSocket::OnSend(nErrorCode);
}
```

После того как функции вставлены, нужно вставить заголовочный файл в диалоговое окно приложения в класс сокета.

```
//MySocket.cpp
...
#include "MySocket.h"
```

После того как требуемые функции событий созданы в классе сокета, вставьте переменную, связанную с нашим классом сокета, в класс диалогового окна. Для сервера нам потребуется две переменные, одна будет связана с прослушиванием запросов на соединение, а другая – с другим приложением. Поскольку существует два объекта сокета, то в диалоговый класс (CSockDlg) введите две переменные. Обе переменные имеют тип класса сокета (CMySocket) и доступ private. Имя одной переменной – m\_sListenSocket, она связана с прослушиванием запроса на соединение, вторая переменная называется m\_sConnectSocket (рис. 8.4) и используется для пересылки сообщения в обоих направлениях.

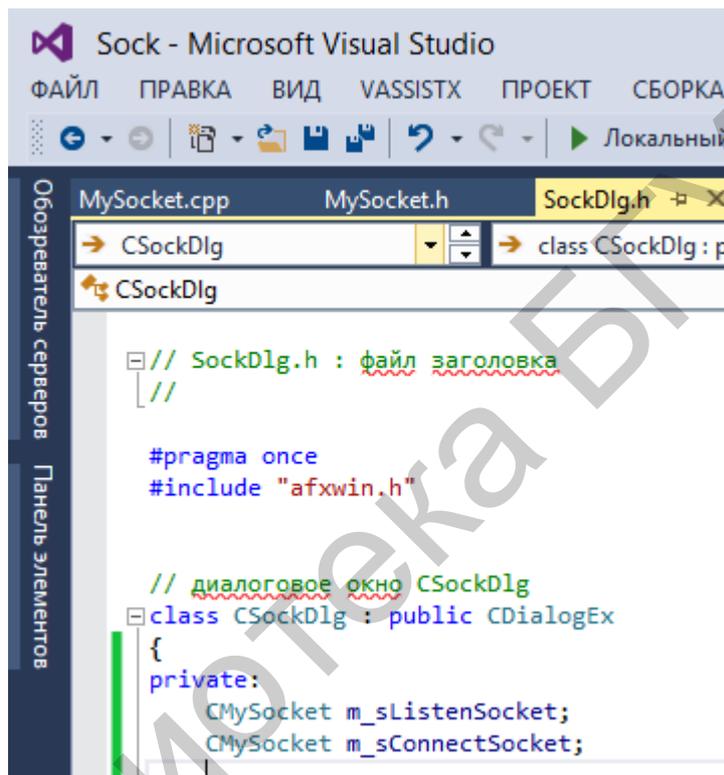


Рис. 8.4. Создание переменных класса CMySocket

После того как переменные сокета созданы, необходимо написать код, инициализирующий эти переменные. По умолчанию задайте тип приложения «клиент», номер порта 4000. Помимо этих параметров установите указатель в объектах сокета, чтобы они указывали на диалоговый класс. Это можно сделать, если вставить код в функцию OnInitDialog.

**Замечание.** Имя, соответствующее loopback, – это специальное имя, используемое в протоколе TCP/IP и обозначающее компьютер, на котором работает приложение. Это внутреннее имя компьютера, превращаемое в адрес 127.0.0.1. Данное имя и адрес компьютера широко используется в тех случаях, когда необходимо осуществить соединение с другим приложением, установленным на том же самом компьютере.

```

BOOL CSockDlg::OnInitDialog()
{
    CDialogEx::OnInitDialog();
    // Инициализируем переменные управления
    m_iType = 0;
    m_strName = "loopback";
    m_iPort = 4000;
    // обновляем элементы управления
    UpdateData(FALSE);
    // Устанавливаем указатель
    m_sConnectSocket.SetParent(this);
    m_sListenSocket.SetParent(this);
    return TRUE;
}

```

Когда пользователь нажимает кнопку, то все функции основного окна становятся недоступными. В этот момент пользователь не может менять параметры программы. Далее происходит обращение к функции `Create` для соответствующей переменной сокета в зависимости от того, используется приложение в виде сервера или в виде клиента. Затем происходит обращение либо к функции `Connect`, либо к функции `Listen`, этим инициализируется соединение с нашей стороны. Чтобы вставить описанные функции в приложение, откройте *Мастер классов* и добавьте функцию для обработки сообщения о событии нажатия `BN_CLICKED` кнопки `Connect` (`IDC_BCONNECT`). Отредактируйте код функции.

```

void CSockDlg::OnVconnect()
{
    // Синхронизируем переменные, используя значения элементов
    // управления
    UpdateData(TRUE);
    // Включаем прочие элементы управления
    GetDlgItem(IDC_BCONNECT)->EnableWindow(FALSE);
    GetDlgItem(IDC_ESERVNAME)->EnableWindow(FALSE);
    GetDlgItem(IDC_ESERVPORT)->EnableWindow(FALSE);
    GetDlgItem(IDC_STATICNAME)->EnableWindow(FALSE);
    GetDlgItem(IDC_STATICPORT)->EnableWindow(FALSE);
    GetDlgItem(IDC_RCLIENT)->EnableWindow(FALSE);
    GetDlgItem(IDC_RSERVER)->EnableWindow(FALSE);
    GetDlgItem(IDC_STATICTYPE)->EnableWindow(FALSE);
    // Работаем в качестве клиента или сервера?
    if (m_iType == 0)
    {
        // клиент, создаем сокет по умолчанию
        m_sConnectSocket.Create();
        // открываем соединение с сервером
        m_sConnectSocket.Connect(m_strName, m_iPort);
    }
    else
    {
        // сервер, создаем возможность прослушивания на указанном порте
        m_sListenSocket.Create(m_iPort);
        // прослушиваем запросы на соединение
    }
}

```

```

        m_sListenSocket.Listen();
    }
}

```

Чтобы завершить приложение, вставьте функции обработки событий сокета OnAccept и OnConnect в диалоговый класс. Эти функции вызываются в классе сокета и не требуют указания каких-либо параметров. Функция OnAccept вызывается в том случае, когда со слушающим сокетом пытается соединиться другое приложение. После того как соединение принято, можно включить окно для ввода текста сообщений, которые наше приложение будет передавать другому.

Чтобы вставить такую функцию в приложение, добавьте новую функцию в диалоговый класс CSocketDlg. Укажите тип функции void, опишите функцию как OnAccept, доступ public. Отредактируйте код функции.

```

void CSocketDlg::OnAccept()
{
    // принимаем запрос на соединение
    m_sListenSocket.Accept(m_sConnectSocket);
    // включаем элементы управления вводимого текста
    GetDlgItem(IDC_EMMSG)->EnableWindow(TRUE);
    GetDlgItem(IDC_BSEND)->EnableWindow(TRUE);
    GetDlgItem(IDC_STATICMSG)->EnableWindow(TRUE);
}

```

На клиентской стороне ничего делать не надо, за исключением включения элементов управления, ответственных за ввод и посылку сообщений. Необходимо также включить кнопку Close, с ее помощью соединение закрывается со стороны клиента (но не сервера). Чтобы добавить в приложение описанные функции, в диалоговый класс CSocketDlg вставьте новую функцию, тип новой функции void, опишите функцию в виде OnConnect, доступ к функции public.

```

void CSocketDlg::OnConnect()
{
    // включаем элементы управления текстом сообщений
    GetDlgItem(IDC_EMMSG)->EnableWindow(TRUE);
    GetDlgItem(IDC_BSEND)->EnableWindow(TRUE);
    GetDlgItem(IDC_STATICMSG)->EnableWindow(TRUE);
    GetDlgItem(IDC_BCLOSE)->EnableWindow(TRUE);
}

```

В диалоговый класс CSocketDlg вставьте три функции, тип всех функций void, а доступ – public. Первая функция – OnSend, вторая – OnReceive, третья – OnClose. Можно скомпилировать приложение.

Запустите две копии приложения. Задайте, чтобы одна из копий работала в режиме сервера, щелкните на кнопке Listen, чтобы перевести его в состояние ожидания запроса на соединение. Все элементы управления при этом будут находиться в отключенном состоянии. Второй экземпляр программы запустите в режиме клиента и нажмите кнопку Connect. Здесь также элементы управления

установлены в выключенное состояние. После того как соединение будет установлено, элементы управления, ответственные за отсылку сообщений, перейдут в рабочее состояние.

**Посылка и прием сообщений.** Необходимо добавить в приложение функции, которые позволили бы осуществлять прием и посылку сообщений. После того как между приложениями установлено соединение, пользователь может ввести текстовое сообщение в окно для редактирования, расположенное в центре диалогового окна, затем, по нажатии кнопки SEND посылается сообщение другому приложению. Чтобы вставить требуемые функции, выполняемые после нажатия кнопки SEND, вначале необходимо позаботиться о том, чтобы была произведена проверка того, содержится ли в окне какое-либо сообщение, затем определить его длину, потом послать сообщение, а затем добавить сообщение в окно списка. Используйте *Мастер классов* для вставки функции, которая будет выполняться после наступления события нажатия кнопки IDC\_BSEND. Отредактируйте функцию.

```
void CSockDlg::OnBsend()
{
    int iLen;
    int iSent;
// Обновляем элементы управления в соответствии с переменными
    UpdateData(TRUE);
    // Есть сообщение для отправки?
    if (m_strMessage != "")
    {
        // Получаем длину сообщения
        iLen = m_strMessage.GetLength();
        // Посылаем сообщение
        char* p = new char[iLen];
        char s1[1024];
        for (int i = 0; i < iLen; i++)
        {
            char s = m_strMessage[i];
            s1[i] = s;
        }
        p = s1;
        iSent = m_sConnectSocket.Send(p, iLen, 0);
        // Смогли послать?
        if (iSent == SOCKET_ERROR)
        {
            return;
        }
        else
        {
            // Добавляем сообщение в список
            m_ctlSent.AddString(m_strMessage);
// Обновляем переменные согласно элементам управления
            UpdateData(FALSE);
        }
    }
}
```

При вызове функции `OnReceive`, что происходит в момент, когда приходит сообщение, оно извлекается из сокета с помощью функции `Receive`. После того как сообщение извлечено, оно преобразуется в тип `String` и добавляется в список полученных сообщений. Эти функции можно создать, если отредактировать существующую функцию `OnReceive` в диалоговом классе.

```
void CSockDlg::OnReceive()
{
    char *pBuf = new char[1025];
    int iBufSize = 1024;
    int iRcvd;
    CString strRcvd;
    // Получаем сообщение
    iRcvd = m_sConnectSocket.Receive(pBuf, iBufSize);
    // Получили что-либо?
    if (iRcvd == SOCKET_ERROR)
    {
        return;
    }
    else
    {
        // Отрезаем конец сообщения
        pBuf[iRcvd] = NULL;
        // Копируем сообщение в CString
        strRcvd = pBuf;
        // добавляем сообщение в список полученных сообщений
        m_ctlRcvd.AddString(strRcvd);
        // обновляем переменные в соответствии с элементами
        // управления
        UpdateData(FALSE);
    }
}
```

**Завершение соединения.** Чтобы закрыть соединение, пользователь клиента может щелкнуть на кнопке `Close`, соединение будет прекращено. В серверном приложении будет получено событие сокета `OnClose`. После этого в обоих приложениях должны произойти одинаковые процессы: соединяющийся сокет должен быть закрыт, элементы управления, ответственные за отсылку сообщений, должны быть выключены. На клиентском приложении, кроме того, необходимо включить элементы управления, ответственные за установку соединения. На серверном приложении процесс прослушивания в ожидании запроса на установление связи восстановится. Чтобы создать необходимые для осуществления описанных действий функции, отредактируйте код функции `OnClose`, изменив код.

```
void CSockDlg::OnClose()
{
    // закрываем сокет
    m_sConnectSocket.Close();
    // выключаем элементы управления, ответственные за посылку
```

```

// сообщений
    GetDlgItem(IDC_EMMSG) ->EnableWindow(FALSE);
    GetDlgItem(IDC_BSEND) ->EnableWindow(FALSE);
    GetDlgItem(IDC_STATICMSG) ->EnableWindow(FALSE);
    GetDlgItem(IDC_BCLOSE) ->EnableWindow(FALSE);
    // мы работаем как клиент?
    if (m_iType == 0)
    {
// да, тогда включаем элементы управления соединением
        GetDlgItem(IDC_BCONNECT) ->EnableWindow(TRUE);
        GetDlgItem(IDC_ESERVNAME) ->EnableWindow(TRUE);
        GetDlgItem(IDC_ESERVPORT) ->EnableWindow(TRUE);
        GetDlgItem(IDC_STATICNAME) ->EnableWindow(TRUE);
        GetDlgItem(IDC_STATICPORT) ->EnableWindow(TRUE);
        GetDlgItem(IDC_RCLIENT) ->EnableWindow(TRUE);
        GetDlgItem(IDC_RSERVER) ->EnableWindow(TRUE);
        GetDlgItem(IDC_STATICTYPE) ->EnableWindow(TRUE);
    }
}

```

Наконец, для кнопки Close необходимо организовать обращение к функции OnClose. Для этого используйте *Мастер классов* и с его помощью вставьте функцию, соответствующую событию нажатия кнопки Close (IDC\_BCLOSE). Отредактируйте код функции.

```

void CSockDlg::OnBclose()
{
    // вызываем функцию OnClose
    OnClose();
}

```

Сейчас после компиляции и запуска двух копий приложения можно осуществить соединение между клиентской и серверной версией приложения и пересылать между ними сообщения в обоих направлениях, а потом разорвать соединение из клиентского приложения, нажав кнопку Close (рис. 8.5).

Можно восстановить соединение клиента с сервером, нажав кнопку Connect еще раз. Если запустить третью копию приложения, изменив в ней номер порта, установив эту копию как сервер, включив режим ожидания запроса на соединение, можно переключать клиентское приложение, поочередно подключаясь то к одному, то к другому серверу, изменяя при этом номер порта.

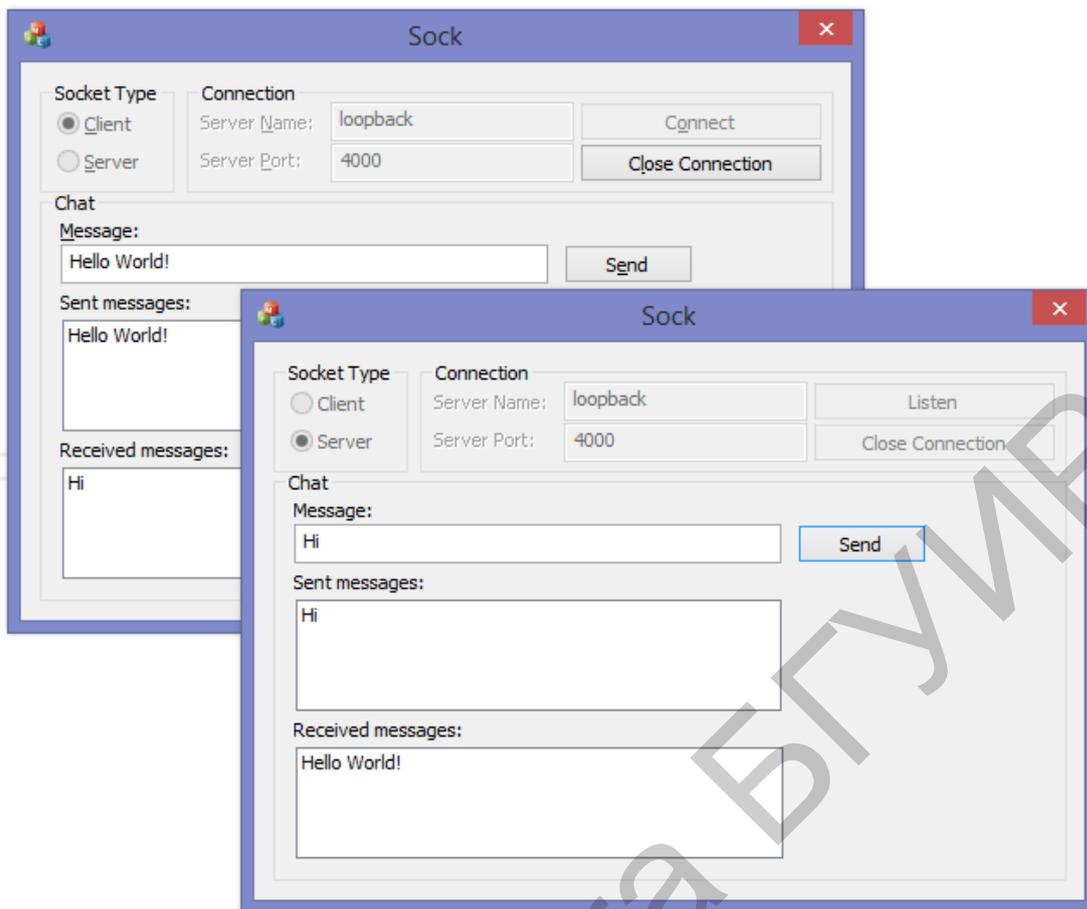


Рис. 8.5. Результат работы приложения

### Задания

1. Разработать интерфейс для обмена сообщениями между пользователями различных узлов сети.
2. Разработать программу организации простых расчетов на сервере для клиентских задач.
3. Организовать взаимодействие, реализующее рассылку сообщений от одного клиента группе клиентов.
4. Организовать пересылку журнала репликаций БД между клиентами Remote-установки через сокеты.

Разработка каркаса приложения с помощью мастера AppWizard

Для разработки каркаса приложения с помощью мастера AppWizard необходимо выбрать **Файл=>Создать=>Проект...** (рис. П.1.1), а затем – вкладку **MFC=>Приложение MFC** в окне **Создать проект**, как это показано на рис. П.1.2.

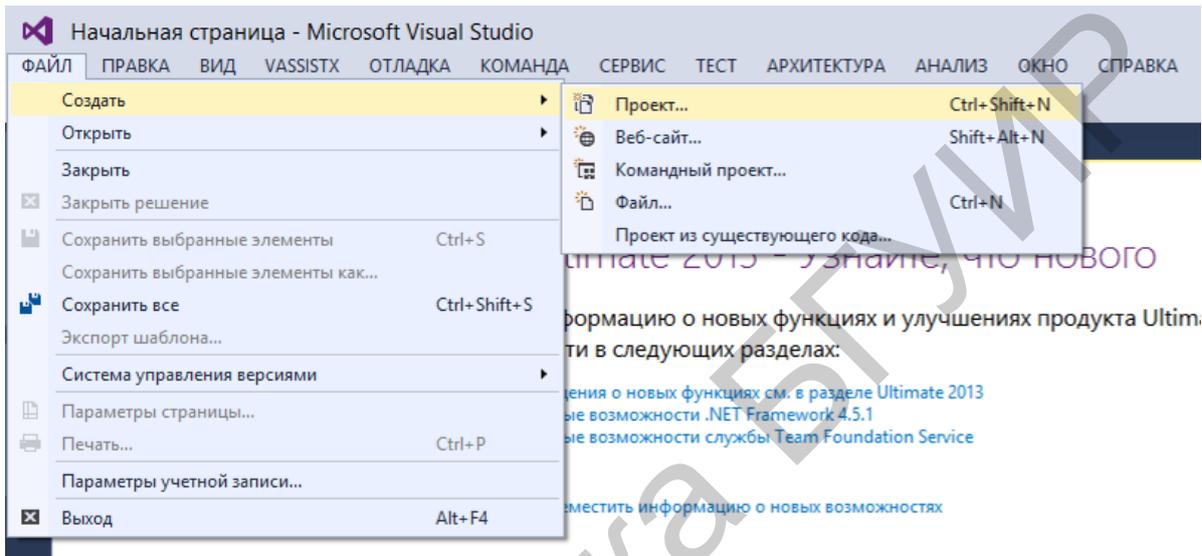


Рис. П.1.1. Начальная страница

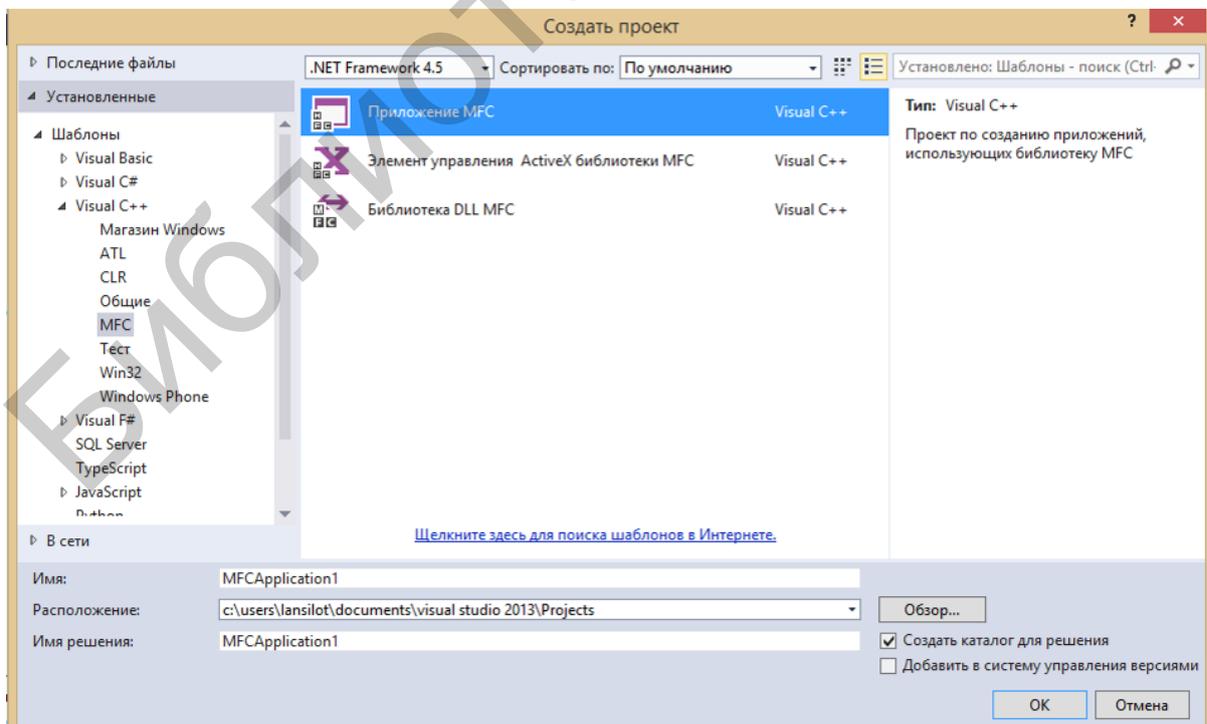


Рис. П.1.2. Выбор типа приложения

В левой части окна находится список возможных типов проектов. Для создания типового приложения необходимо выбрать **Приложение MFC**. Также необходимо указать имя проекта в поле **Имя**, а в поле **Расположение** – каталог, в котором будет находиться проект. Дальнейшие действия будем называть «шагами» или «этапами». Данный тип проекта использует библиотеку классов Microsoft Foundation Classes (MFC). На каждом этапе программист может изменить некоторые параметры создаваемого приложения. Для перехода на следующий этап необходимо щелкнуть на кнопке **Далее**, для перехода к предыдущему этапу – щелкнуть на кнопке **Назад**. При нажатии кнопки **Отмена** процесс создания приложения вернется к этапу **Выбор типа** приложения (см. рис. П.1.2). Кнопка **Готово** позволяет завершить сеанс настройки, пропустив последующие этапы и настроить все оставшиеся параметры в состоянии по умолчанию. Рассмотрим более подробно этапы создания приложения:

**1. Тип приложения.** Первое, что должен определить программист, приступая к работе в MFC Application Wizard, – сколько документов будет поддерживать будущее приложение, т. е. будет ли оно MDI-приложением, SDI-приложением или простым диалоговым приложением. Для каждого из этих типов приложений Application Wizard создает различные классы. Окно MFC Application Wizard при этом будет выглядеть так, как показано на рис. П.1.3.

SDI-приложение (SDI – Single Document Interface, интерфейс с единственным документом) позволяет в каждый момент времени иметь открытым только один документ. Создание SDI-приложения настраивается в окне MFC Application Wizard переключателем **Один документ**.

MDI-приложение (MDI – Multiple Document Interface, многодокументный интерфейс) может одновременно держать открытыми несколько документов, каждый из которых представлен отдельным файлом, например: Excel, Word. Создание MDI-приложения настраивается в окне MFC AppWizard переключателем **Несколько документов** (Multiple documents).

Простое диалоговое приложение, как правило, вообще не открывает документов. Примером может служить приложение Character Map (Таблица символов). Такие приложения не имеют меню. Создание приложения этого типа настраивается в окне MFC Application Wizard переключателем **На основе диалоговых окон** (Dialog based).

Ниже этой группы переключателей в диалоговом окне находится флажок **Поддержка архитектуры документ/представление** (Document/View architecture support). Если не будет оговорено отдельно, будем считать, что флажок **Поддержка архитектуры документ/представление** должен быть установлен.

Остальные параметры в этом окне следует оставить по умолчанию, если не оговорено отдельно.

**2. Поддержка составных документов (Compound Document Support).** Второй этап создания приложения – выбор уровня поддержки операции с

составными документами. Окно MFC Application Wizard при этом будет выглядеть так, как показано на рис. П.1.4.

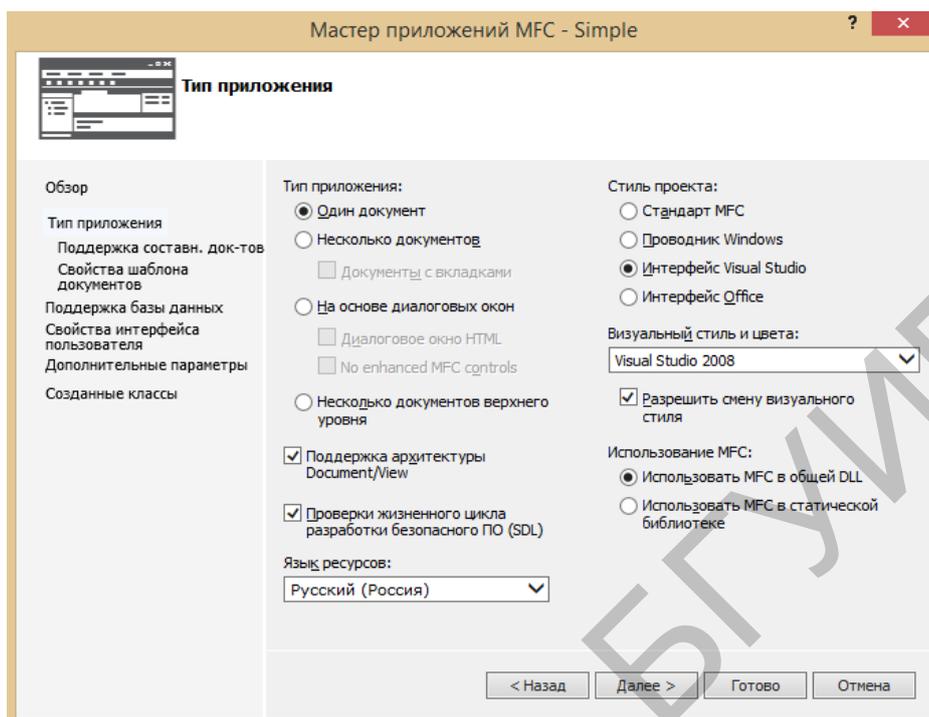


Рис. П.1.3. Выбор варианта интерфейса пользователя

На выбор предлагается пять вариантов поддержки. Если не планируется создание OLE-приложения, выберите переключатель **Никакой** (None). Если вы хотите, чтобы в приложении использовались связанные или внедренные объекты OLE (например, такие, как документы Word или рабочие листы Excel), выберите переключатель **Контейнер** (Container). Если планируется создание приложения, документы которого могли бы быть внедрены в другое приложение, но при этом само приложение не будет использоваться автономно, выберите переключатель **Мини-сервер** (Mini server). Если ваше будущее приложение будет не только служить сервером для других приложений, но и сможет работать автономно, выберите переключатель **Полный сервер** (Full server). Если создаваемое приложение должно обладать способностью включать документы других приложений и само обслуживать их своими объектами, выбирайте переключатель **Контейнер** или **Полный сервер** (Container/Full server).

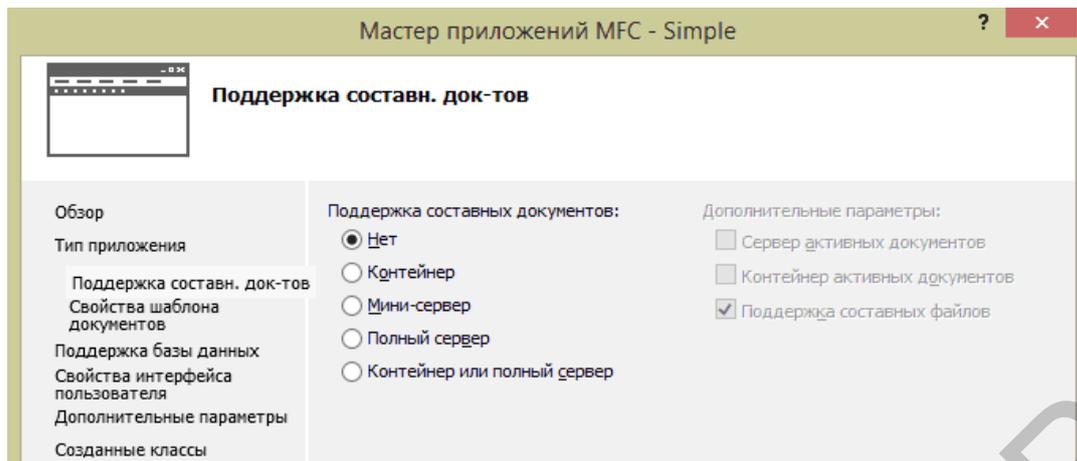


Рис. П.1.4. Выбор варианта поддержки составных документов

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. В диалоговом окне имеется группа переключателей: **Поддержка составных файлов** (Support for compound files), **Сервер активных документов** (Active document server) и **Контейнер активных документов** (Active document container).

**3. Свойства шаблона документов (Document Template Properties)** оставим без изменений, нажав *Далее*.

**4. Поддержка базы данных (Database Support).** Окно *Мастера приложений* MFC при этом будет выглядеть так, как показано на рис. П.1.5. Здесь на выбор предлагаются четыре варианта уровня поддержки базы данных. Если работа с базами данных в приложении не планируется, выберите переключатель **Нет** (None). Если предполагается иметь доступ к базам данных, но для этого не будут использованы классы просмотра, производные от CFormView, или нет необходимости в меню **Запись** (Record), выберите переключатель **Только файлы заголовка** (Header files only). Если вы планируете разрабатывать классы просмотра базы данных в приложении как производные от CFormView и иметь меню **Запись** (Record), но не нуждаетесь в средствах сохранения-восстановления (сериализации) документов, выбирайте переключатель **Представление базы данных без поддержки файлов** (Database view without file support). Если помимо всего, что задано в предыдущем случае, вы планируете и сохранение-восстановление документов на диске, выберите **Представление базы данных с поддержкой файлов** (Database view with file support).

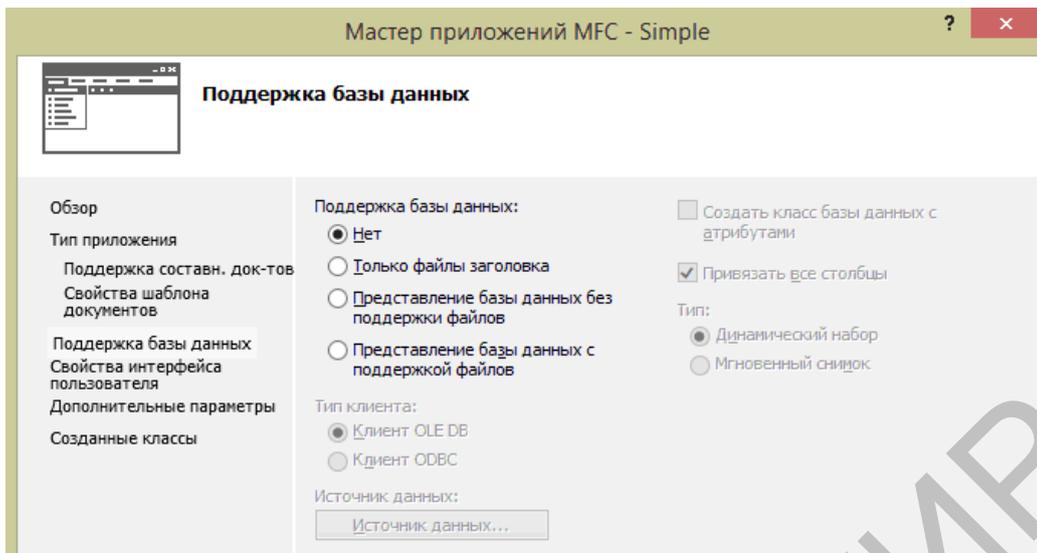


Рис. П.1.5. Выбор варианта поддержки базы данных

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

**5. Свойства интерфейса пользователя (User Interface Features).** Следующий этап – выбор опций, определяющих внешний вид элементов пользовательского интерфейса. **Мастер приложений MFC** при этом будет выглядеть так, как показано на рис. П.1.6. Диалоговое окно содержит много переключателей-флажков, соответствующих предлагаемым опциям оформления.

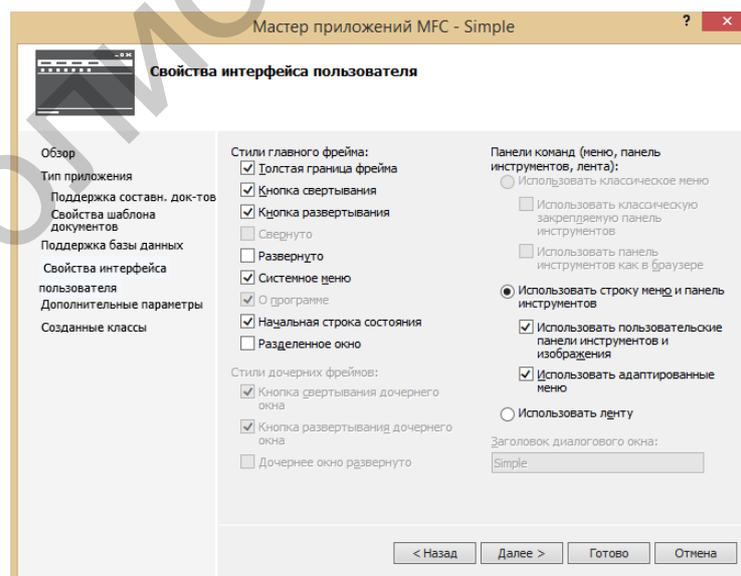


Рис. П.1.6. Установка некоторых опций пользовательского интерфейса

**6. Дополнительные параметры (Advanced Feature).** На рис. П.1.7 представлена настройка дополнительных параметров приложения.

При установке опции **Интерфейс MAPI** (Messaging API) приложение сможет обмениваться сообщениями по электронной почте. При установке опции **Сокеты Windows** приложение сможет иметь непосредственный доступ к Интернету через такие протоколы, как FTP и HTTP.

Если вы хотите, чтобы создаваемое приложение было программируемым, установите флажок **Автоматизация** (Automation). Если планируется использовать в приложении элементы управления ActiveX, установите флажок **Элементы ActiveX**.

Можно установить длину списка последних открываемых файлов в поле меню **Файл** создаваемого приложения. Для этого служит раскрывающийся список **Число файлов в текущем списке файлов**, по умолчанию этот параметр имеет значение 4 и менять его не рекомендуется без очень веских причин.

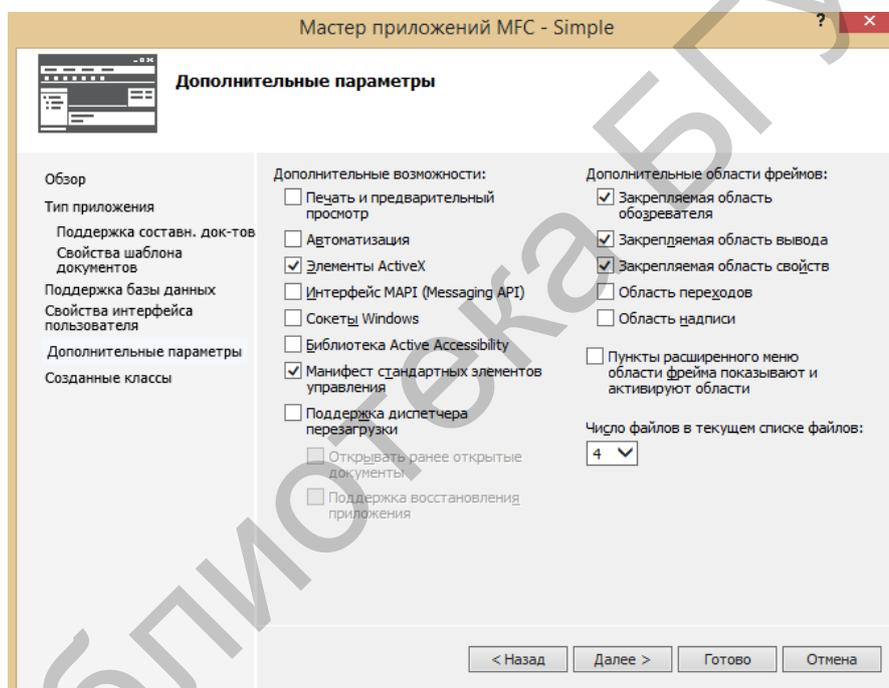


Рис. П.1.7. Установка дополнительных параметров

**7. Созданные классы (Generated Classes).** Последний этап создания приложения – подтверждение имен классов и имен файлов, как это показано на рис. П.1.8. **Мастер приложений** использует имя проекта (в данном случае – Simple) для формирования имен классов и имен файлов. Нет необходимости 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 кнопку **Готово**.

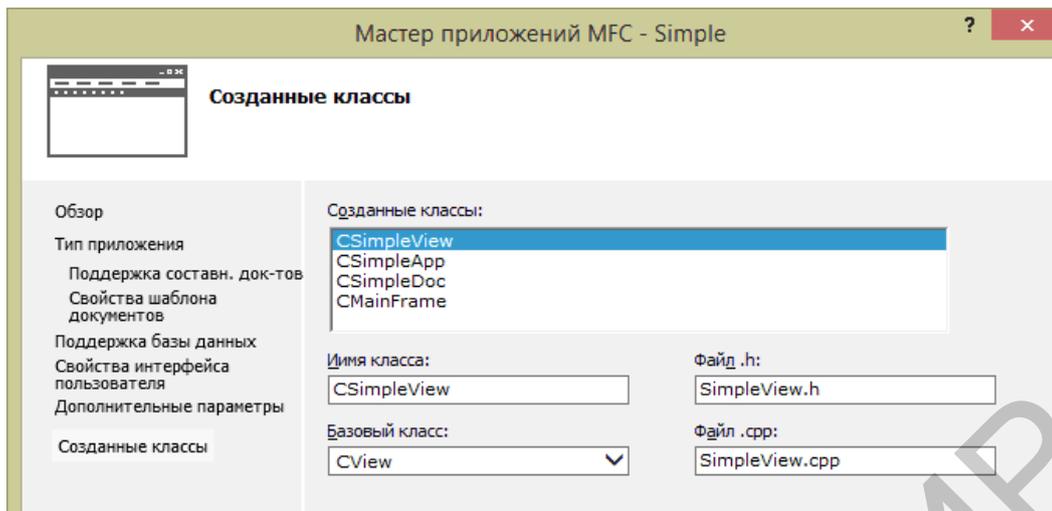


Рис. П.1.8. Подтверждение имен классов и файлов

## Диалоговые окна

Приложение может иметь любое количество диалоговых окон, в которых происходит ввод данных пользователем. Как правило, для каждого диалогового окна в приложении существуют ресурс диалога и класс. В класс окна включены переменные и функции-члены, ответственные за работу диалога.

Ресурсы диалога создаются посредством редактора ресурсов, с помощью которого возможно включать в его состав необходимые элементы управления и размещать их в необходимом порядке. Класс создается при помощи **Мастера классов**. Как правило, класс диалогового окна в проекте является производным от класса `CDialog`, входящего в MFC. **Мастер классов** также позволяет облегчить работу с элементами управления, расположенными на диалоговом окне. 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 или считать значения элементов управления, необходимо обращаться к членам-переменным класса.

**Формирование нового ресурса диалогового окна.** Первый шаг процесса организации диалогового окна в приложении – формирование ресурса окна. Чтобы приступить к формированию ресурсов, необходимо открыть **Обозреватель решений** и открыть `Simple.rc` двойным щелчком левой кнопки мыши. Выбрать папку `Dialog` и в контекстном меню выбрать **Добавить ресурс...** (рис. П.2.1).

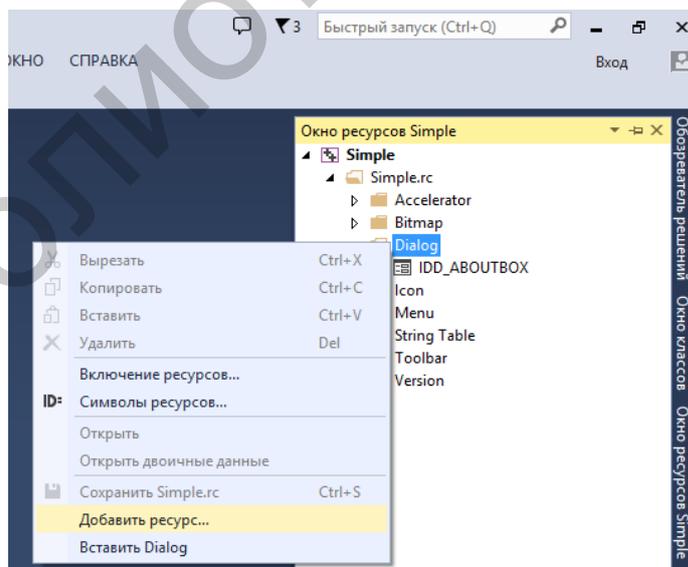


Рис. П.2.1. Добавление ресурса

В результате отобразится окно *Добавление ресурса*. Выберите элемент Dialog и нажмите кнопку *Создать* (рис. П.2.2).

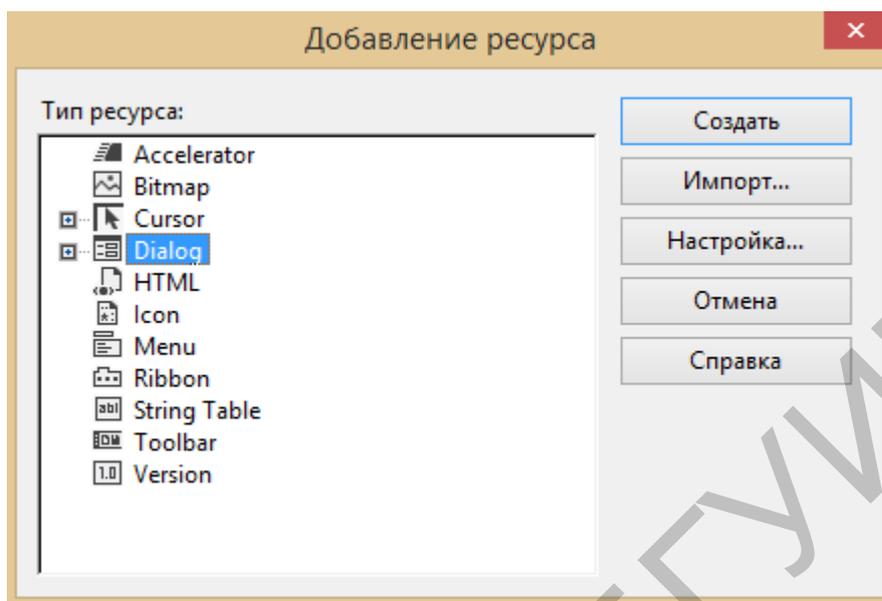


Рис. П.2.2. Добавление ресурса (диалогового окна) Dialog

В результате откроется редактор диалогового окна, который выводит на экран заготовку окна, как это показано на рис. П.2.3.

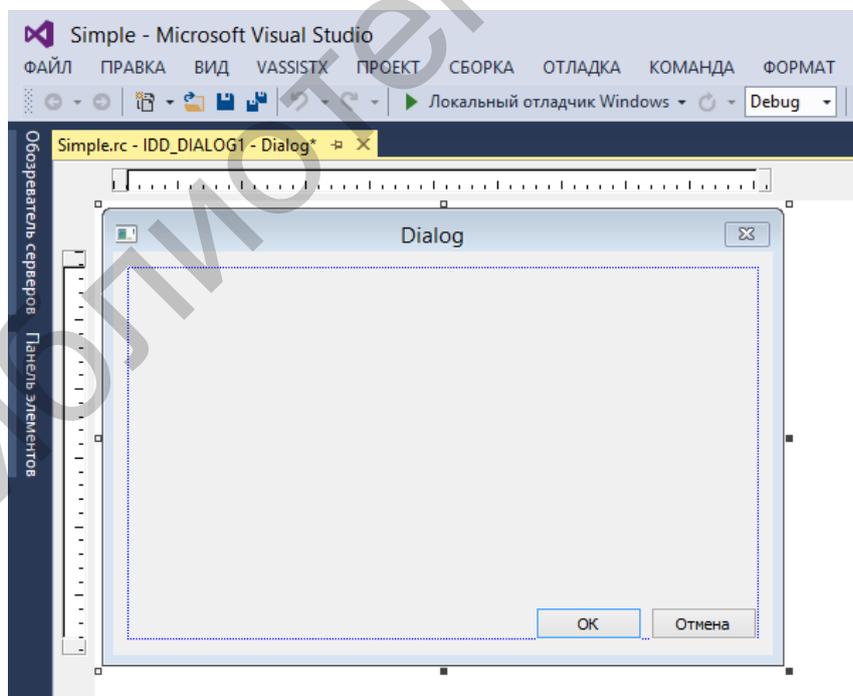


Рис. П.2.3. Заготовка диалогового окна

Поле **Свойства** для создаваемого диалогового окна отображается, если навести указатель на диалоговое окно, щелкнуть правой кнопкой мыши и выбрать пункт меню **Свойства**. В поле **Надпись** (Caption) введите заголовок (например, Simple) диалога, как это показано на рис. П.2.4.

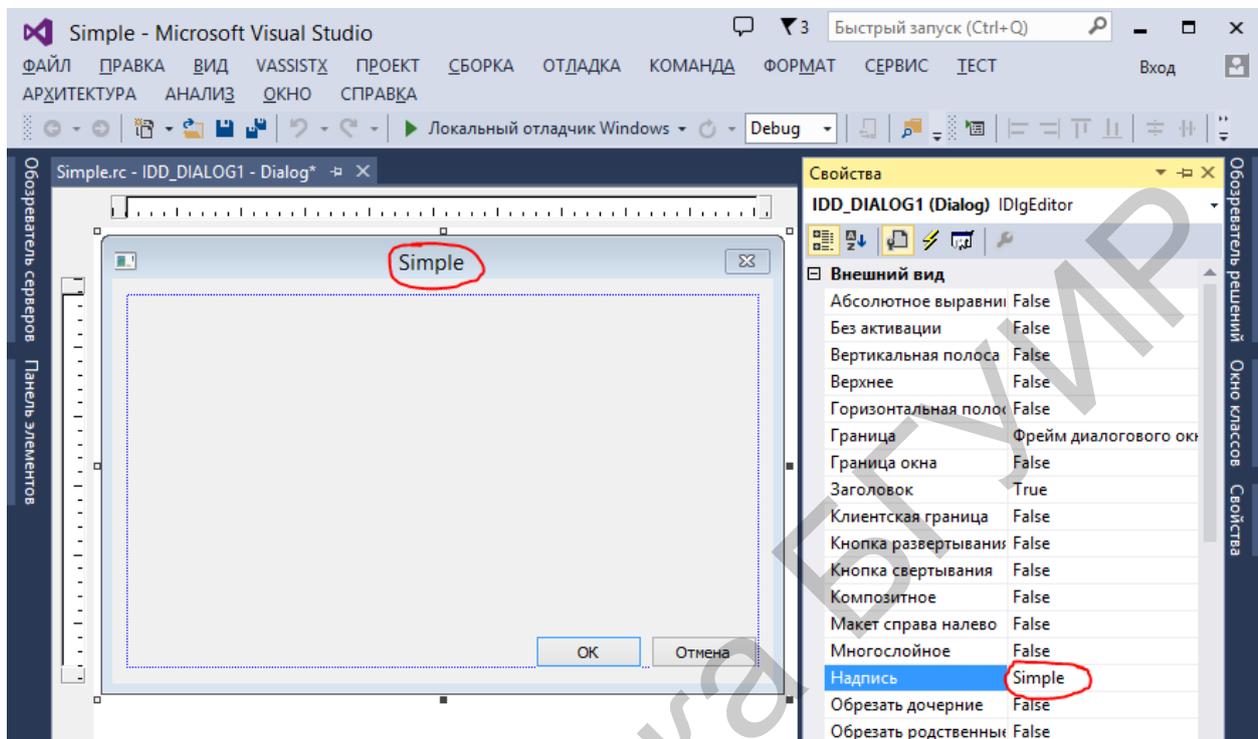


Рис. П.2.4. Свойства (Properties) создаваемого диалогового окна

**Создание класса диалогового окна.** Когда формирование ресурсов диалогового окна будет завершено, дважды щелкните левой кнопкой мыши, наведя указатель на окно, или выберите пункт меню **Добавить класс...**, щелкнув правой кнопкой мыши. Таким образом, вы вызовете на экран диалоговое окно **Мастер добавления классов**. **Мастер добавления классов** обнаружит новый диалог и предложит создать новый класс. Появится диалоговое окно, которое показано на рис. П.2.5. В поле **Имя класса** введите имя нового класса (например, CSimpleDlg) и щелкните на **Готово**. После этого **Мастер добавления классов** создаст новый класс, подготовит файл текста программы SimpleDlg.cpp и файл заголовка SimpleDlg.h и включит их в состав проекта.

**Модальные и немодальные диалоговые окна.** 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 **Файл=>Открыть** 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 *Правка=>Поиск* и *Правка=>Замена* во многих приложениях Windows.

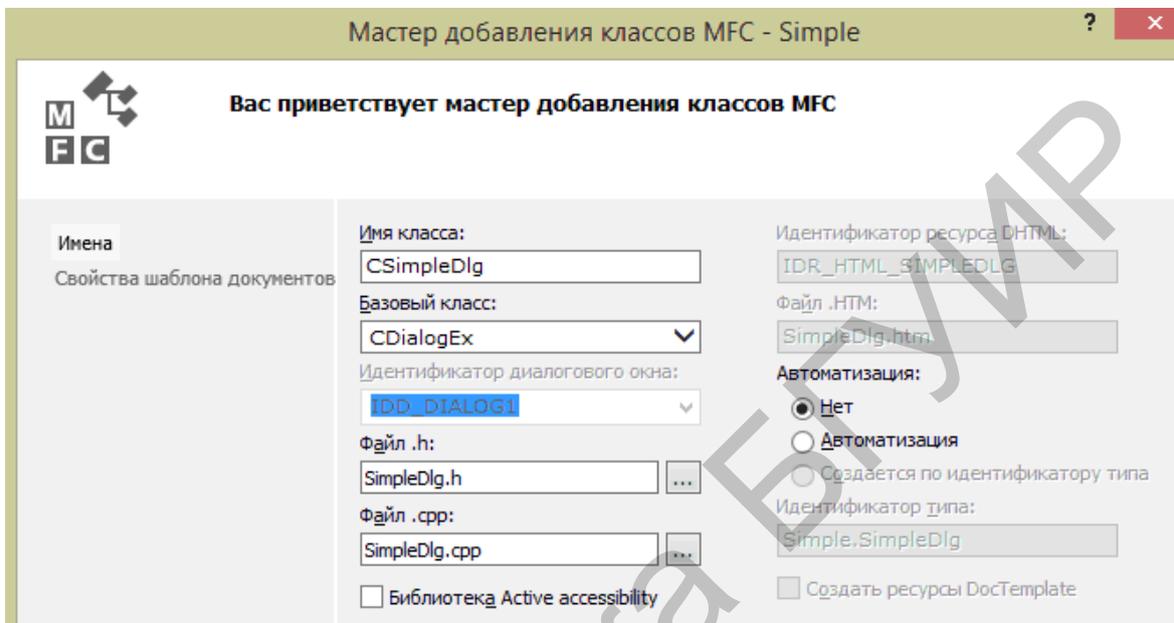


Рис. П.2.5. Создание класса для диалогового окна

## Меню

*Мастер приложений* формирует в заготовке для MDI-приложения два меню (для SDI одно). Если ни один файл не открыт в приложении, выводится меню IDR\_MAINFRAME; если же хотя бы один документ открыт, выводится меню IDR\_MDITYPE. Обратите внимание, что меню IDR\_MAINFRAME не имеет пунктов (выпадающих меню) **Правка** и **Окна**, а само меню **Файл** короче, чем в IDR\_MDITYPE. В первом варианте в нем есть только пункты **Создать**, **Открыть**, **Сохранить**, **Сохранить как...** (**Печатать**, **Предварительная печать**, **Настройка принтера**), **Последний файл** и **Выход** (рис. П.3.1).

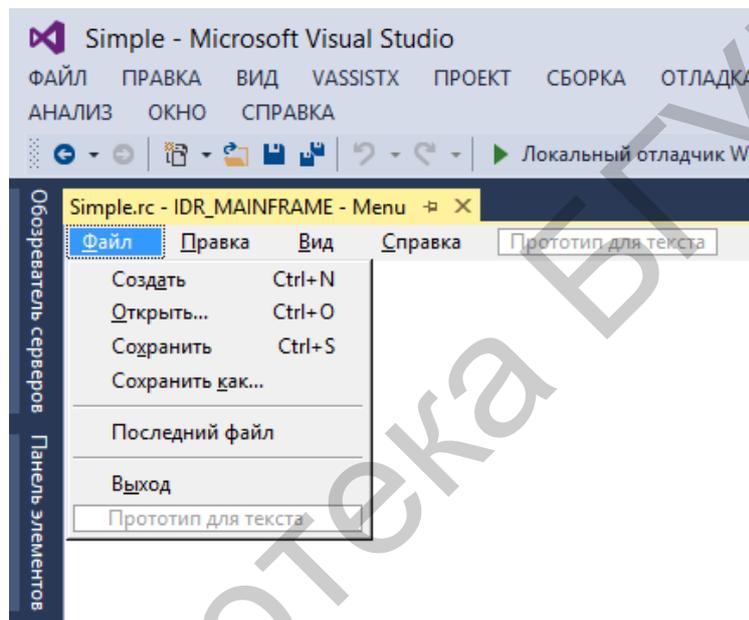


Рис. П.3.1. Редактирование ресурса типа меню

Выберите из списка ресурсов меню. Появляется изображение меню. Необходимо выбрать пункт, в который необходимо добавить новую команду. В конце дочерних пунктов есть свободное место для нового пункта. В поле **Прототип для текста** вводится текст, который будет отображаться при выводе меню на экран.

Щелкнув правой кнопкой мыши на новом пункте меню, можно вызвать **Свойства**, где в поле **Надпись** можно изменить название этого пункта. Каждый пункт меню имеет свой идентификатор, посредством которого программа связывается с ним. Данный идентификатор вводится в поле **ID** диалога **Свойства**. Если оставить это поле пустым, то Visual Studio сама автоматически назначит уникальный идентификатор (рис. П.3.2).

Для создания и подключения клавиш-акселераторов к пунктам меню (и другим элементам управления) необходимо развернуть узел Accelerator в дереве ресурсов и выбрать необходимую таблицу акселераторов.

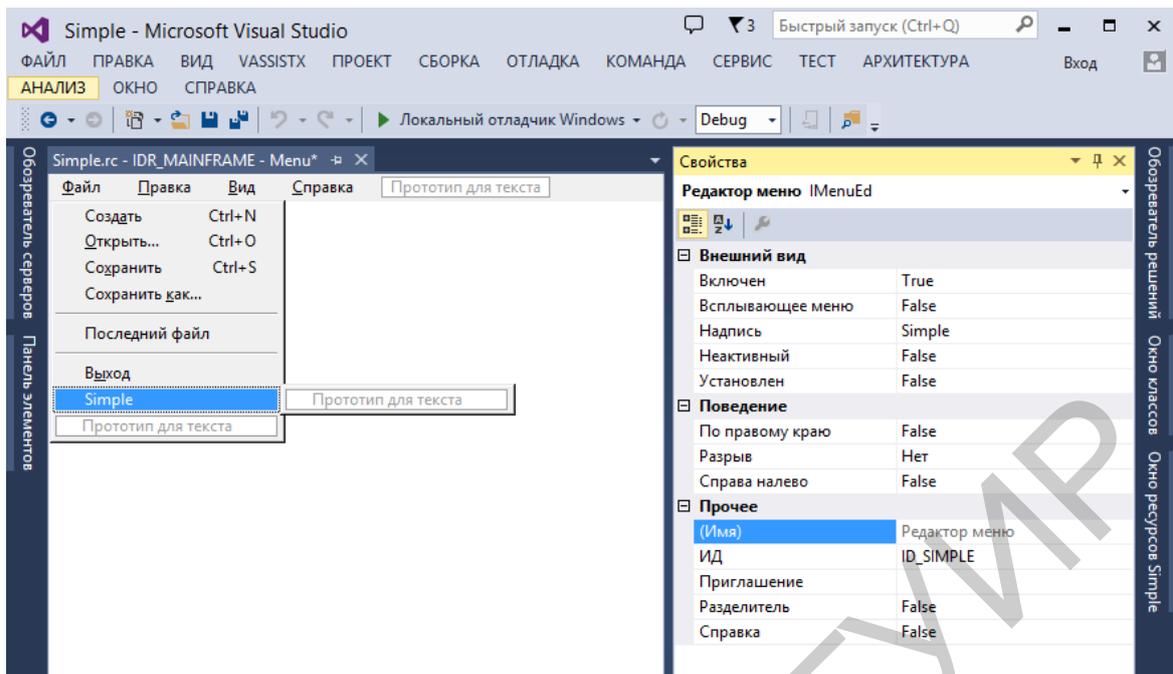


Рис. П.3.2. Добавление нового пункта меню

Далее требуется найти пустую строку в таблице и выбрать пункт **Свойства** в выпадающем меню. В поле **ИД** необходимо выбрать идентификатор требуемого пункта меню. В поле **Клавиша** вводится символ, который будет использоваться в акселераторе. Также необходимо указать клавиши-модификаторы (Ctrl, Alt, Shift), которые тоже используются в акселераторе (рис. П.3.3).

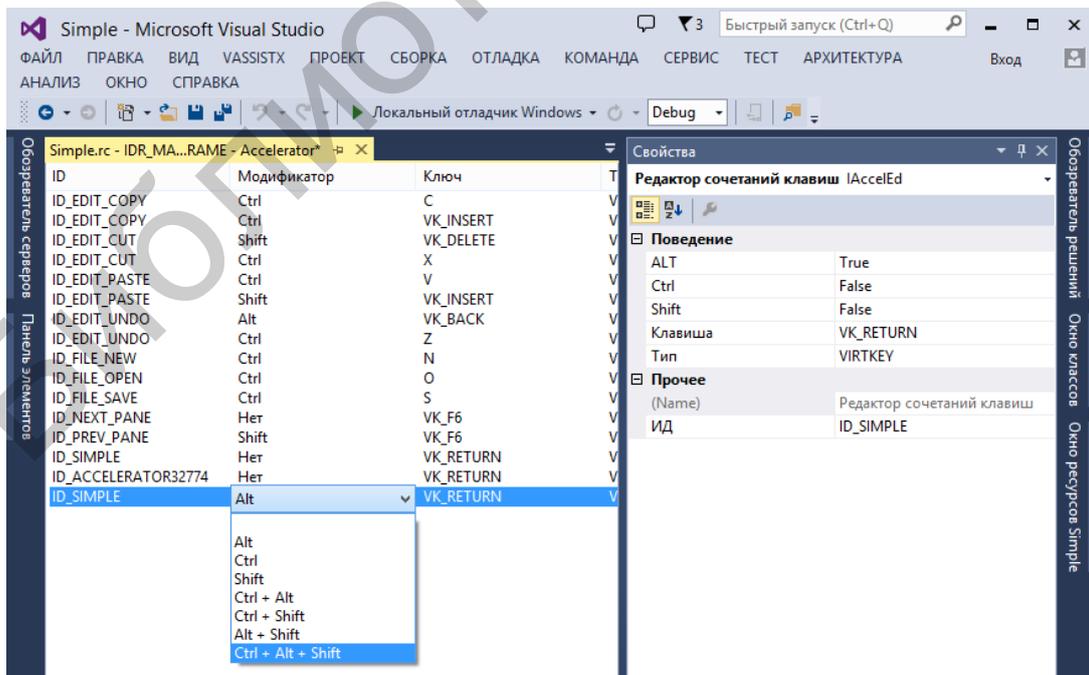


Рис. П.3.3. Добавление новой клавиши-акселератора

## Панель инструментов

Генерируемая *Мастером приложений* панель инструментов (Toolbar) содержит пиктограммы для наиболее распространенных команд. Однако при разработке часто требуется удалить или добавить пиктограммы. Для работы с панелью инструментов необходимо развернуть узел ToolBar в дереве ресурсов и выбрать необходимую панель – откроется окно редактирования панели инструментов (рис. П.4.1).

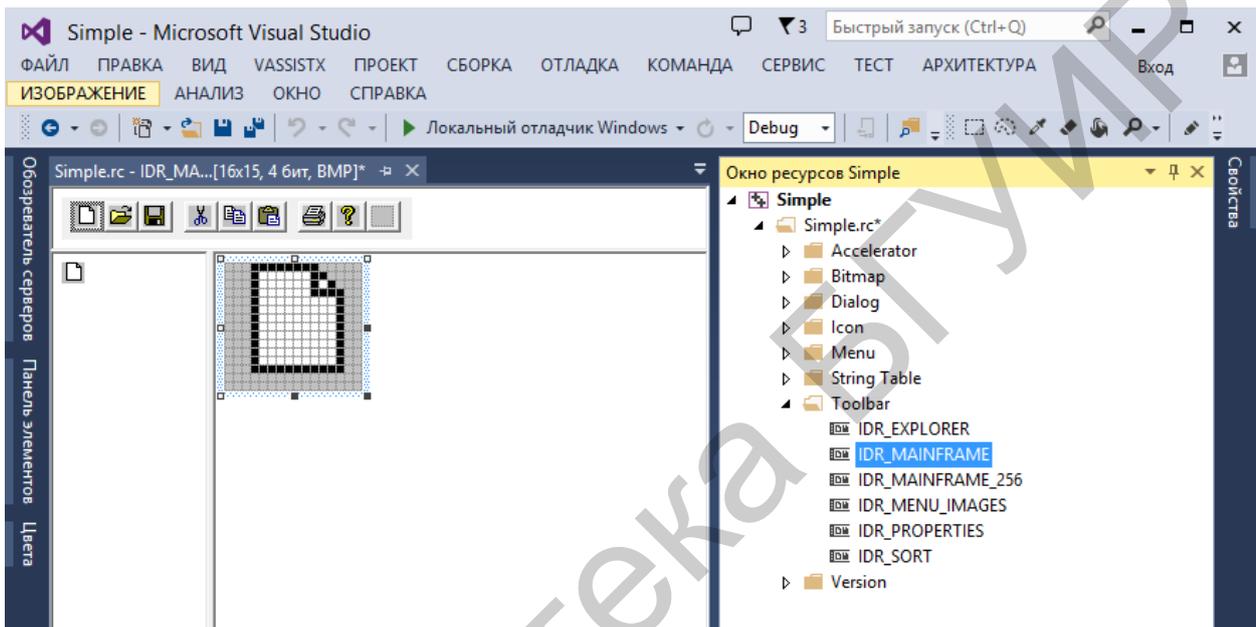


Рис. П.4.1. Редактирование панели инструментов

После того как окно редактора панелей инструментов будет открыто, удаление пиктограмм с панели инструментов сводится к простому перетаскиванию их с панели на свободное место в окне.

**Добавление пиктограмм на панель инструментов.** Процедура включения новой пиктограммы в панель инструментов состоит из двух этапов. На первом из них следует нарисовать изображение пиктограммы, а на втором необходимо связать команду с новой пиктограммой. Приступая к созданию изображения новой пиктограммы, необходимо выбрать заглушку пустой пиктограммы, расположенной на формируемой панели инструментов. Изображение пустой пиктограммы в увеличенном масштабе появится в окне редактирования.

Далее выведите на экран окно свойств и назначьте пиктограмме соответствующий идентификатор команды (рис. П.4.2).

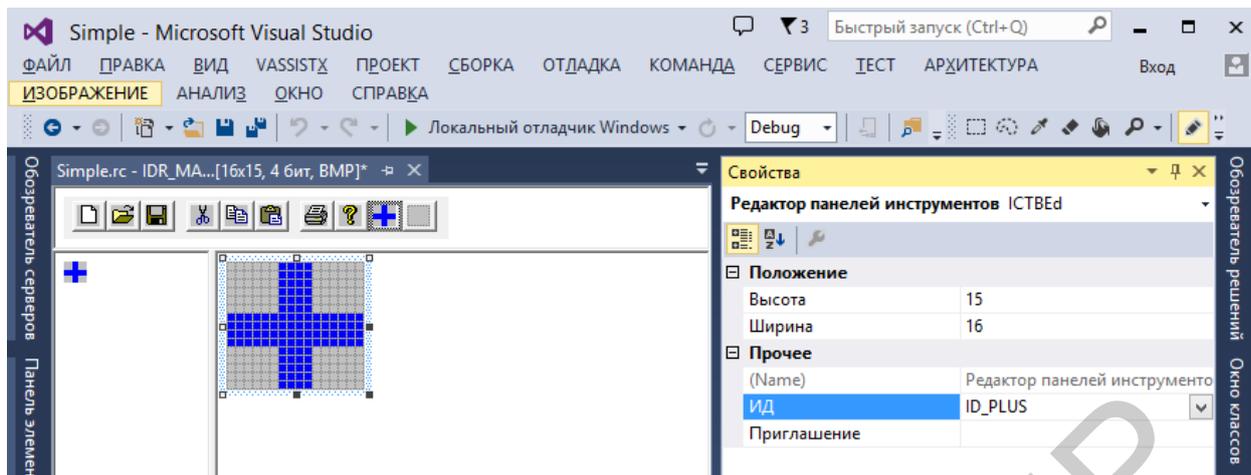


Рис. П.4.2. Определение свойств пиктограммы

Библиотека БГУИР

Генерация приложения, связанного с базой данных

Процесс создания приложения для работы с базой данных отличается от процесса создания простого приложения лишь выбором на четвертом этапе **Мастера приложений** опции **Поддержка базы данных**.

В данном диалоговом окне необходимо выбрать либо **Представление базы данных без поддержки файлов** (Database view without file support), если нет необходимости работать с другими файлами, либо **Представление базы данных с поддержкой файлов** (Database view with file support) (рис. П.5.1). На данном этапе необходимо определить способ доступа к данным (ODBC или OLE DB). Для определения источника данных необходимо нажать кнопку **Источник данных** – появится диалоговое окно, после чего в соответствующем поле нужно выбрать источник данных (рис. П.5.2).

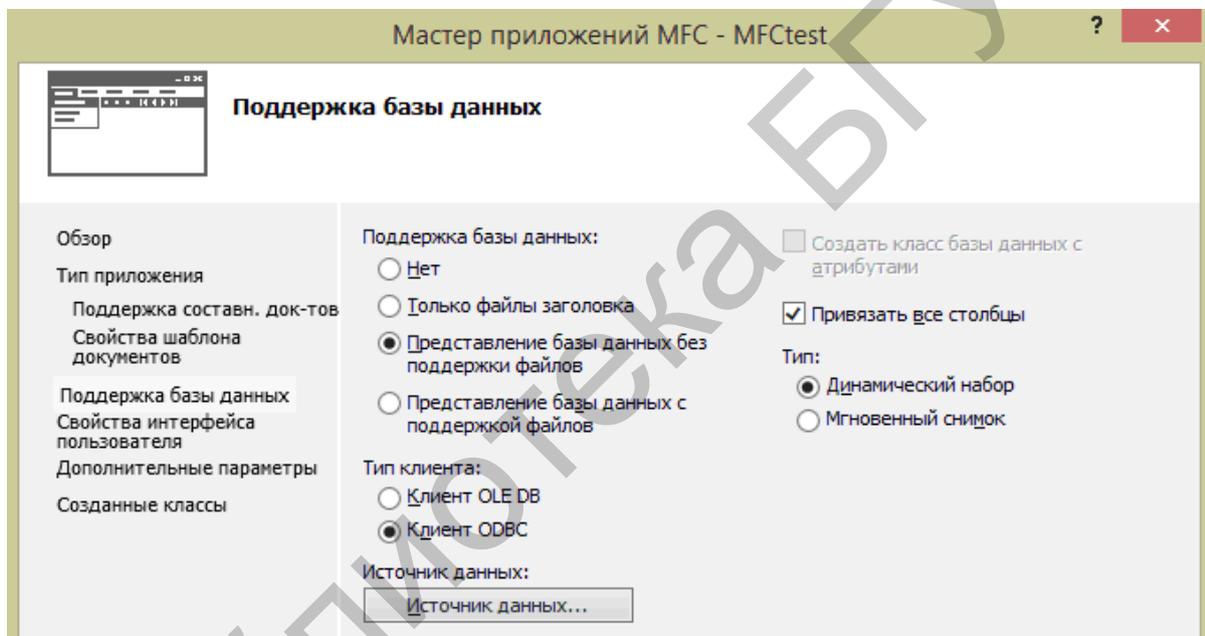


Рис. П.5.1. Диалог поддержки работы проекта с базой данных

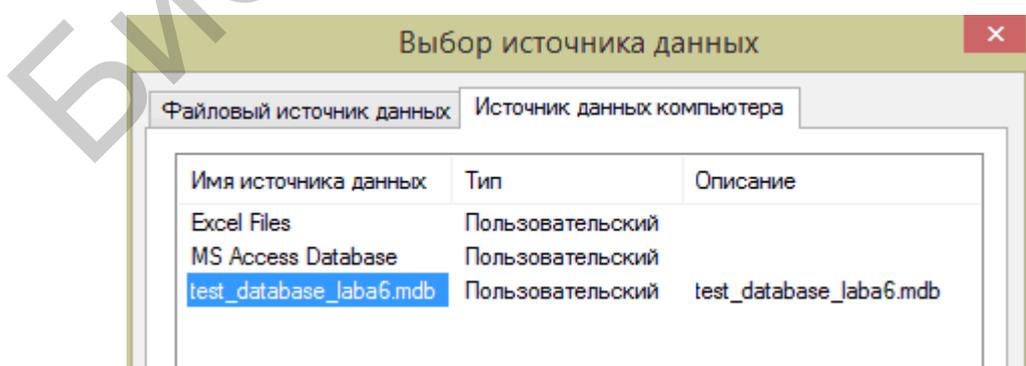


Рис. П.5.2. Выбор источника данных компьютера

После окончания работы с диалогом необходимо выбрать таблицы, данные которых будут доступны программе (рис. П.5.3). После выбора источника данных и таблиц процесс выбора возвращается ко второму этапу *Мастера приложений*.

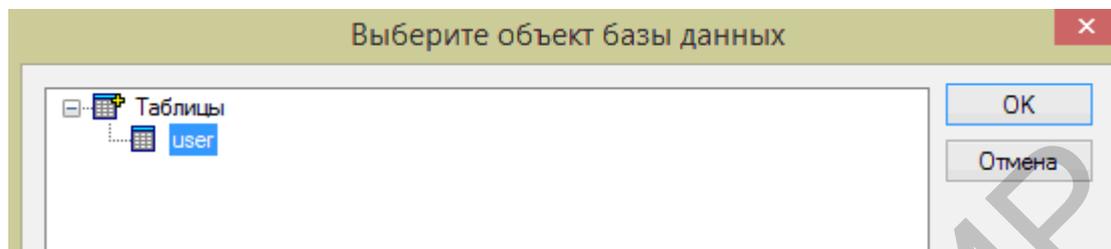


Рис. П.5.3. Выбор таблицы user

## Список использованных источников

1. Кириенко, Н. А. Разработка Windows-приложений на языке С++ с использованием библиотеки MFC : учеб.-метод. пособие / Н. А. Кириенко. – Минск : БГУИР, 2012. – 202 с.
2. Визуальные средства разработки приложений : учеб.-метод. пособие / В. Н. Комличенко [и др.]. – Минск : БГУИР, 2004. – 68 с.
3. 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. – 88 с.
4. Хортон, А. Visual С++ 2010. Полный курс / А. Хортон ; пер. с англ. – М. : Издательский дом «Вильямс», 2011. – 1216 с.
5. Сидорина, Т. Л. Самоучитель Microsoft Visual Studio С++ и MFC / Т. Л. Сидорина. – СПб. : БХВ, 2009. – 843 с.
6. Зиборов, В. В. MS Visual С++ 2010 в среде .NET. Библиотека программиста / В. В. Зиборов. – СПб. : БХВ, 2012. – 320 с.
7. Голощапов, А. Л. Microsoft Visual Studio 2010 / А. Л. Голощапов. – СПб. : БХВ, 2011. – 544 с.
8. Культин, Н. Б. Программирование в Visual С++ 2010. Самоучитель / Н. Б. Культин. – СПб. : БХВ, 2012. – 384 с.

*Учебное издание*

**Кириенко** Наталья Алексеевна  
**Живицкая** Елена Николаевна  
**Комличенко** Виталий Николаевич

**ВИЗУАЛЬНЫЕ СРЕДСТВА РАЗРАБОТКИ  
ПРОГРАММНЫХ ПРИЛОЖЕНИЙ.  
ЛАБОРАТОРНЫЙ ПРАКТИКУМ**

ПОСОБИЕ

Редактор *Е. С. Юрец*  
Корректор *Е. И. Герман*  
Компьютерная правка, оригинал-макет *Е. Д. Стенусь*

Подписано в печать 27.03.2018. Формат 60x84 1/16. Бумага офсетная. Гарнитура «Таймс».  
Отпечатано на ризографе. Усл. печ. л. 7,79. Уч.-изд. л. 8,0. Тираж 80 экз. Заказ 341.

Издатель и полиграфическое исполнение: учреждение образования  
«Белорусский государственный университет информатики и радиоэлектроники».  
Свидетельство о государственной регистрации издателя, изготовителя,  
распространителя печатных изданий №1/238 от 24.03.2014,  
№2/113 от 07.04.2014, №3/615 от 07.04.2014.  
ЛП №02330/264 от 14.04.2014.  
220013, Минск, П. Бровки, 6