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

Кафедра экономической информатики

ВИЗУАЛЬНЫЕ СРЕДСТВА РАЗРАБОТКИ ПРИЛОЖЕНИЙ

Учебно-методическое пособие
по курсу
«Объектно-ориентированное проектирование и программирование»
для студентов специальности 40 01 02-02
«Информационные системы и технологии в экономике»
дневной формы обучения

Минск 2004

УДК 004.414.2 (075.8)
ББК 32.973.26-018 я 73
В 42

Рецензент:
доцент кафедры интеллектуальных информационных технологий БГУИР,
кандидат физ.-мат. наук, доцент Н.А. Гулякина

Авторы:
В. Н. Комличенко, О.П. Едемская, Н.А. Кириенко, Е.Н. Унучек,
Б.Л. Лутович, В.М. Комар

В 42 **Визуальные** средства разработки приложений: Учеб.-метод. пособие по курсу «Объектно-ориентированное проектирование и программирование» для студентов спец. 40 01 02-02 «Информационные системы и технологии в экономике» дневной формы обучения / В. Н. Комличенко, О.П. Едемская, Н.А. Кириенко и др. – Мн.: БГУИР, 2004.– 68 с.: ил.
ISBN 985-444-549-6.

В учебно-методическом пособии представлен курс из четырех лабораторных работ, даны краткие теоретические сведения, необходимые для их выполнения, а также примеры.

УДК 004.414.2 (075.8)
ББК 32.973.26-018 я 73

ISBN 985-444-549-6

Содержание

Лабораторная работа № 1

Использование технологии OLE DB

Цель работы:

1. Узнать назначение и возможности технологии OLE DB.
2. Ознакомиться с процессом создания простого приложения доступа к источнику данных с использованием технологии OLE DB.
3. Научиться использовать механизм OLE DB для организации работы с источником данных.

Первоначально OLE была задумана как технология интеграции программных продуктов, входящих в комплект Microsoft Office. Предшественницей OLE является реализованная в Windows технология динамического обмена данными DDE (Dynamic Data Exchange), до сих пор широко применяемая в этой среде. Однако многие разработчики не без оснований считают, что DDE трудно использовать, поскольку это технология низкого уровня. По существу, DDE представляет собой модель взаимодействия процессов – протокол, с помощью которого приложение может организовать канал обмена данными с DDE-сервером, находящимся на той же машине. DDE – это асинхронный протокол. Иными словами, после установления связи вызывающая сторона передает запрос и ожидает возврата результатов. Такой механизм более сложен, чем синхронный вызов функции, так как нужно учитывать вероятность нарушения связи, тайм-ауты и другие ошибки, которые приложение должно распознавать и исправлять. Низкая популярность DDE вынуждала Microsoft искать различные способы его усовершенствования.

Для решения этой проблемы архитекторы OLE создали группу технологий, область применения которых гораздо шире составных документов (DDE). Основу OLE 2 составляет важнейшая из этих технологий — Модель многокомпонентных объектов (Component Object Model — COM). Новая версия OLE не только обеспечивает поддержку составных документов лучше, чем первая, но и, несомненно, идет куда дальше простого объединения документов, созданных в разных приложениях. OLE 2 позволяет по-новому взглянуть на взаимодействие любых типов программ.

Новые возможности программных систем многим обязаны COM, предоставившей общую парадигму взаимодействия программ любых типов: библиотек, приложений, системного программного обеспечения и др. Вот почему подход, предложенный COM, можно использовать при реализации практически любой программной технологии, и его применение дает немало существенных преимуществ.

1.1.Наиболее важные технологии COM

Автоматизация

Электронные таблицы, текстовые процессоры и другие программы предоставляют все виды полезных возможностей. Чтобы реализовать эти возможности, приложения должны предоставлять свои сервисы не только человеку, но и программам — они должны быть программируемыми. Обеспечение программируемости и является целью автоматизации (Automation, первоначально называвшейся OLE-автоматизацией).

Приложение можно сделать программируемым, обеспечив доступ к его сервисам через обычный COM-интерфейс. Однако так поступают редко. Вместо этого доступ к сервисам приложений осуществляется через диспинтерфейсы (dispinterface). Они очень похожи на интерфейсы (у них есть методы, клиенты осуществляют к ним доступ через указатель интерфейса и т. д.).

Программируемый доступ к внутренним сервисам посредством автоматизации поддерживается и рядом других приложений. Именно эта возможность легкого доступа к мощным средствам существующих приложений делает автоматизацию одной из наиболее широко используемых технологий на основе COM.

Перманентность

Объекты состоят из методов и данных, и многим объектам необходимо сохранять свои данные в течение периодов неактивности. Объекту нужно сделать свои данные перманентными (persistent), что обычно означает запись их на диск. COM-объекты достигают этого разными путями. Один из наиболее широко применяемых — структурированное хранилище (Structured Storage).

Традиционные файловые системы обеспечивают совместное использование приложениями одного дискового устройства без конфликтов между ними. Каждое приложение работает со своими собственными файлами и, может быть, даже с собственными подкаталогами, независимо от того, чем заняты в тот же момент другие приложения. Приложениям не требуется взаимодействовать друг с другом, чтобы сохранить свои данные, так как у каждого есть отдельная область для хранения.

Так как COM обеспечивает совместную работу разных типов программ с помощью одной модели, то независимо разработанный COM-объект может стать частью чего-то, что пользователь будет считать одним приложением, и в то же время объекту по-прежнему будет необходимо хранить свои данные на диске отдельно, поэтому используется способ совместного использования одного файла несколькими COM-объектами. Такую возможность и предоставляет структурированное хранилище. Создавая, по сути дела, файловую систему внутри каждого файла, структурированное хранилище предоставляет каждому компоненту, составляющему некоторое приложение, собственный отдельный кусок пространства хранилища, собственные “файлы”. С точки зрения пользователя файл только один. Однако с точки зрения приложения каждый компонент имеет собственную область для хранения данных, и все такие области находятся внутри одного дискового файла.

Чтобы реализовать все это, структурированное хранилище определяет два типа COM-объектов, каждый из которых поддерживает соответствующие интерфейсы. Эти объекты известны как хранилища (storage) и потоки (streams) и аналогичны соответственно каталогам и файлам обычной файловой системы. Файл структурированного хранилища может содержать данные многих COM-объектов, каждый из которых использует для сохранения своих данных собственное хранилище или поток. Точно так же, как обычная файловая система обеспечивает совместное использование диска несколькими приложениями, структурированное хранилище позволяет разным приложениям сообща использовать один файл.

Моникеры

Моникер (moniker, имя, кличка*)* сам по себе является COM-объектом, но весьма специфического назначения: любой моникер знает, как создать и инициализировать экземпляр другого объекта. Например, имея моникер для банковского счета, можно попросить его создать счет, инициализировать его и соединить с ним. Все детали, необходимые для выполнения этих действий, скрыты от клиента. Если он хочет работать посредством моникеров с двумя банковскими счетами, то ему потребуется два отдельных моникера, по одному для каждого объекта — счета. Вообще моникеры в среде COM не необходимы; они просто облегчают жизнь клиентов.

1.2. Единообразная передача данных и объекты с подключением

Стандартный способ обмена информацией в мире COM — единообразная передача данных (Uniform Data Transfer). Как и любая технология OLE, использующие его приложения должны поддерживать определенные интерфейсы COM. Методы этих интерфейсов определяют стандартные способы для описания передаваемых данных, для указания их местоположения и собственно для их пересылки. Они даже определяют простой механизм, позволяющий одному приложению уведомить другое о том, что нужные последнему данные стали доступны. Хотя единообразная передача данных вряд ли является наиболее достойным внимания аспектом COM, она играет важную роль в работе COM-приложений.

Полезная в определенных ситуациях простая схема, определенная как единообразная передача данных для уведомления клиента о наличии интересующих его данных, не вполне достаточна. Именно для ликвидации этих недостатков на основе COM была разработана технология объектов с подключением (Connectable Objects). Обеспечивая более общий механизм обратной связи объекта с клиентом, объекты с подключением позволяют клиенту легко получать уведомления об интересующих его событиях.

Составные документы

В текстовые процессоры добавляются графические возможности, в электронные таблицы – средства построения диаграмм, и, кажется, все кончится созданием одного большого приложения для решения всех задач. Но в действительности цель как раз не в этом, а в интеграции разных приложений.

Например, добавлять поддержку графики в текстовый процессор не потребуется, если внутри него можно будет использовать некоторое уже существующее графическое приложение. Пользователю должно представляться нечто такое, что выглядит как один документ, хотя на самом деле над разными частями такого документа совместно работают разные приложения.

Для решения этой проблемы предназначена технология OLE (ранее известная как документы OLE — OLE Documents). Поддерживая нужные COM-объекты, каждый с собственным набором интерфейсов, независимые приложения могут совместно работать, чтобы пользователь получил один составной документ. Все эти интерфейсы носят абсолютно общий характер — ни одно приложение не знает, что представляют собой другие. OLE поможет просто задействовать в случае необходимости существующее приложение электронной таблицы.

Определенный OLE стандартный интерфейс обеспечивает взаимодействие между приложениями любых типов и любых производителей, а не только между электронными таблицами и текстовыми процессорами Microsoft.

При создании составного документа с помощью OLE одно из приложений всегда является ***контейнером***. Как следует из названия, контейнер определяет самый общий документ, в котором содержится все остальное. Другие приложения-***серверы*** -могут размещать свои документы внутри документа-контейнера.

При использовании OLE документ сервера может быть либо ***связан***, либо ***внедрен*** в документ контейнера. Связанный документ сервера хранится в отдельном файле, а в документе контейнера хранится лишь связь с этим файлом. (На самом деле связью является моникер.) Внедренный документ сервера хранится в том же файле, что и документ контейнера. (Два приложения при этом совместно используют общий файл с помощью структурированного хранилища).

Для стандартизации доступа к различным базам данных Microsoft предлагает средства различного уровня, такие как открытая связь с базами данных (ODBC), объекты доступа к данным (DAO), удалённые объекты данных (RDO). Компоненты доступа к данным Microsoft (MDAC) представляют собой набор инструментов основных технологий, используемых Microsoft для универсального доступа к данным, описание которых приведено ниже:

- ODBC представляет собой проверенный интерфейс API, позволяющий приложению получить доступ к реляционной информации из различных систем управления базами данных.
- OLE DB представляет собой открытую спецификацию набора системных интерфейсов низкого уровня, базирующихся на COM и разработанных источниках информации. Все специфические особенности источника информации скрыты в интерфейсе OLE DB и представлены в общем формате, с которым приложение работает одним и тем же образом.
- ADO представляет собой открытую спецификацию набора системных интерфейсов уровня приложения, базирующихся на COM и созданных для поддержки разработки. ADO использует OLE DB для доступа к данным и предоставляет разработчику, знакомому с DAO и RDO, удобный интерфейс.

OLE DB определяет открытый, расширяемый набор интерфейсов, которые выделяют и инкапсулируют независимые части функциональности СУБД

(контейнеры рядов, процессоры запросов и координаторы транзакций) обеспечивающие унифицированный доступ к разнообразным источникам информации. В свою очередь функциональность OLE DB включает доступ и обновление данных, обработку запросов, уведомления, транзакции, защиту и удалённый доступ к данным. Определяя унифицированный набор интерфейсов доступа к данным, компоненты OLE DB не только способствуют унификации доступа к разным источникам информации, но и позволяют уменьшить требования приложений к объёму памяти, позволяя им задействовать только те возможности СУБД, которые действительно необходимы.

1.3. Методические указания

Для построения данного приложения, выполним приведенную ниже последовательность действий. Создадим заготовку программы по шаблону, как указано в приложении 5 лабораторного практикума «Визуальные средства разработки приложений» [2], с небольшими изменениями: для выбора OLE DB в качестве объекта доступа к данным на шаге 2 в диалоге *Database Options* в группе переключателей *Data Source* выберем переключатель *OLE DB*, после чего кнопка с надписью *Select OLE DB Data source* станет активной. Теперь нажмём на эту кнопку. В результате появится окошко *Data Link Properties (*рис.1.1).

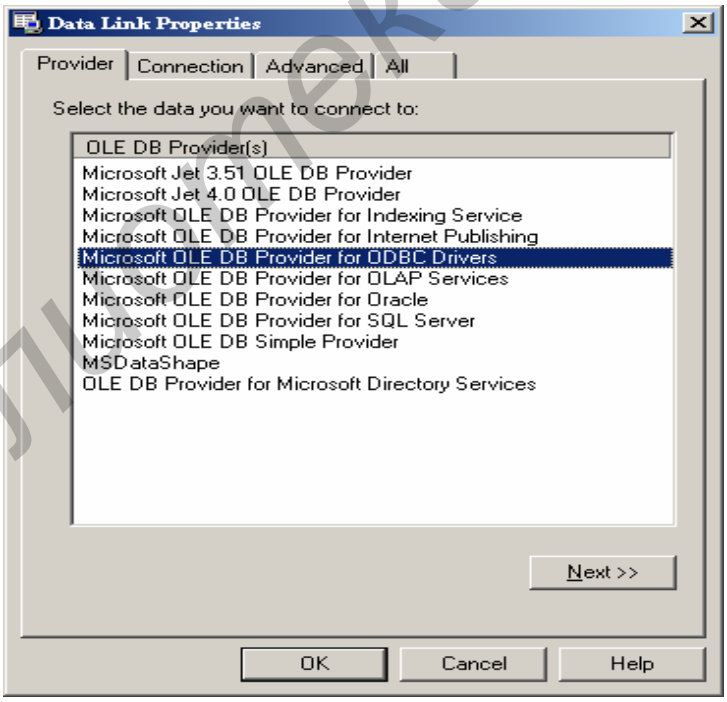

Рис.1.1. Диалог выбора провайдера ODBC драйвера

Затем нажимаем кнопку *Next,* что переводит нас на закладку *Connection.* Теперь в элементе *"Use data source name"* выбираем источник данных (в нашем примере это база данных с именем *test*)(рис. 1.2).

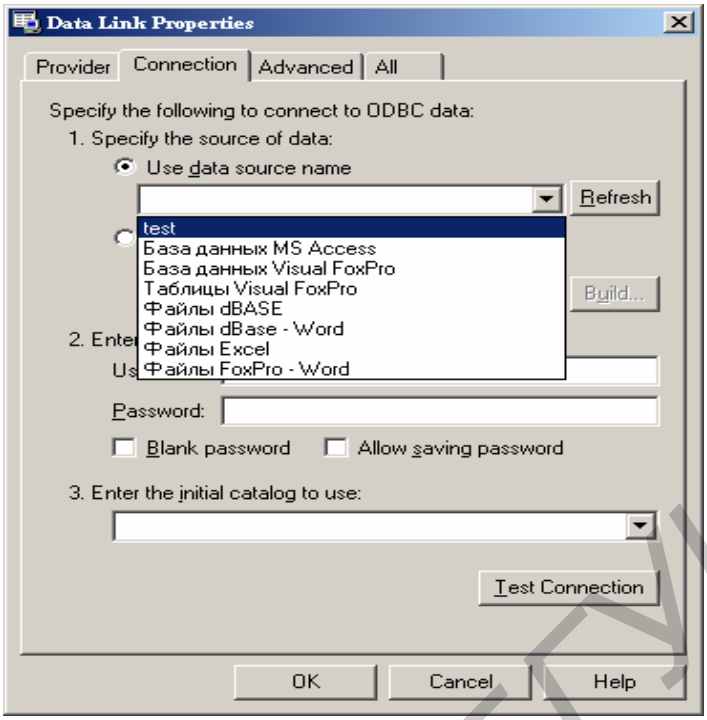

Рис.1.2. Выбор источника данных

После этого в п. 3 *"Enter the initial catalog to use*" выбираем абсолютный путь к источнику данных. Затем нажимаем кнопку *"Test Connection"*. В результате этого при успешном соединении с источником данных на экран выводится сообщение (рис. 1.3).

Рис. 1.3. Сообщение об успешном тестировании подсоединения к источнику данных

Для разграничения доступа при обращении к источнику данных необходимо перейти на закладку *"Advanced"*, где в графе *"Access permissions"* следует выбрать соответствующие права доступа. В нашем случае выберем: *Read/Write (Чтение/Запись),* как показано на рис. 1.4.

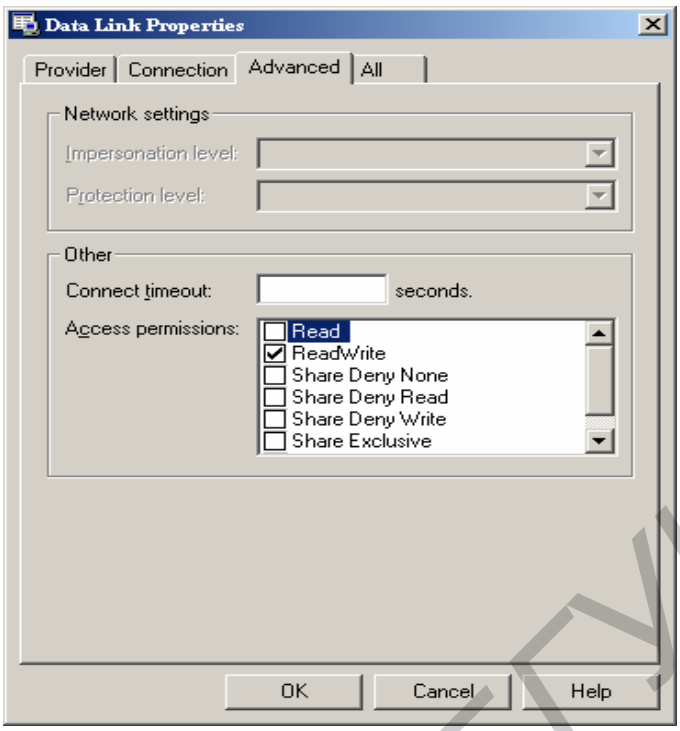

Рис.1.4. Назначение прав доступа на работу с источником данных

После нажатия на кнопку *"Ok"* переходим вновь к диалогу *"Database options"*. После подтверждения, теперь уже в диалоге *"Database options"*, на экране появляется диалог *"Select Database Table"*, в котором мы выбираем имя таблицы, с которой хотим работать (в нашем примере это *test_table*) (рис.1.5).

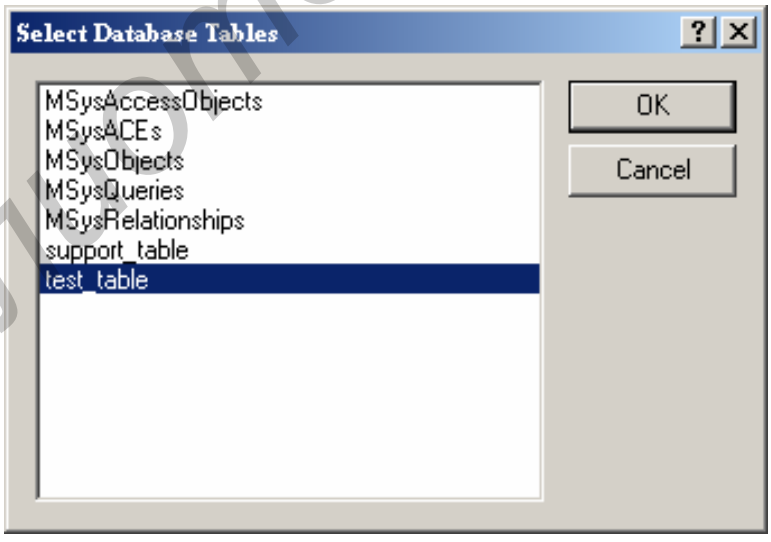

Рис. 1.5. Диалог выбора таблицы в источнике данных

Последующие шаги можно сразу же пропустить, нажав кнопку *"Finish"*.

После этого, мастер AppWizard автоматически генерирует классы, содержащие методы, с помощью которых мы можем работать с нашей базой данных.

Теперь займёмся построением графического интерфейса. Расположим элементы управления типа Edit Box и Button, так как показано на рис.1.6.

Рис. 1.6. Расположение элементов управления на форме

Зададим размеры окна нашего приложения в классе *CMainFrame* (функция *PreCreateWindow)*. Для этого перед строкой *"return TRUE;"* добавляем запись из двух строк:

```
cs.cx = 315;
cs.cy = 195;
```

Теперь для позиционирования курсора в источнике данных в классе *CMy_oledbView* создадим метод *displayMembersOfDataSourse()*

```
oid COledbView::displayMembersOfDataSourse()
{
CString str;
str.Format("%d ",m_pSet->m_id_table); //форматирование
   //визуальное отображение данных
m_edit_id.SetWindowText(str);
str = m_pSet->m_name_table;
m_edit_name.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 EditBox содержимое текущей позиции курсора источника данных. *m_pSet* является указателем, генерируемым *AppWizard*, на объект класса *COledbSet*, через который реализуется связь нашего приложения с базой данных.

Теперь в *ClassWizard* необходимо создать переменные на наши элементы управления типа EditBox. После добавления закладка *"Member Variables"* в диалоге *ClassWizard* должна выглядеть, как показано на рис. 1.7.

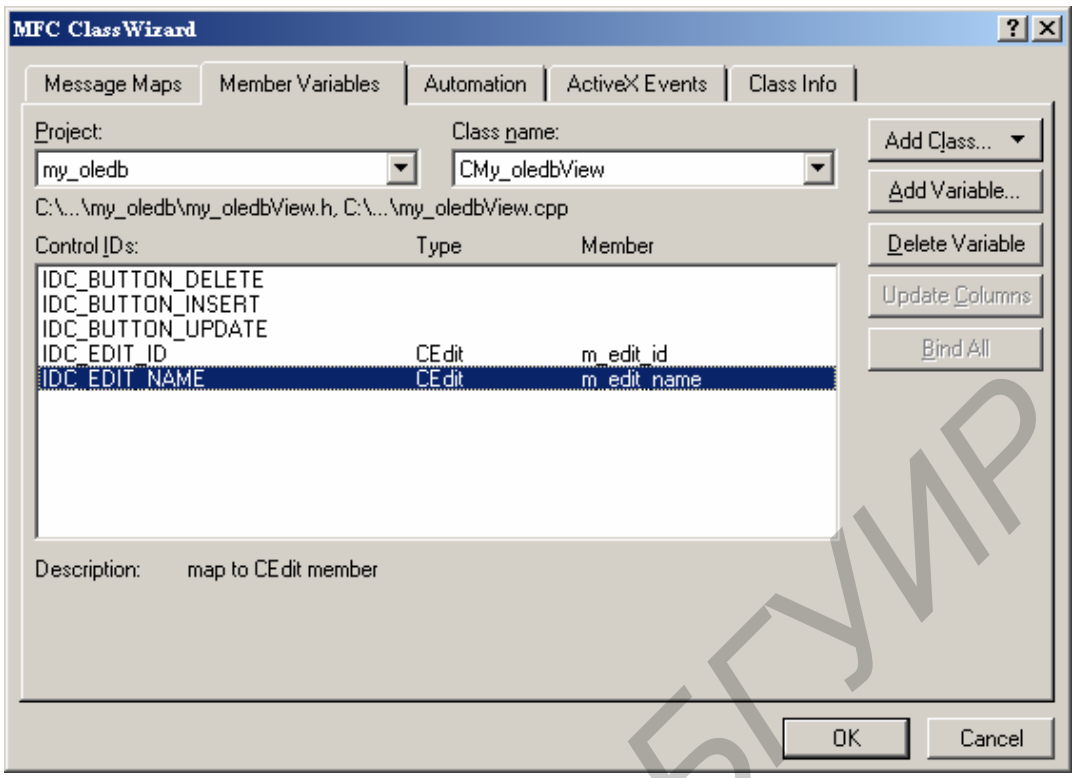

Рис. 1.7. Вид закладки “Member Variables” после создания переменных на элементы управления

Теперь перейдём к описанию модулей для работы с курсором источника данных.

Для этого необходимо переопределить стандартные методы, используемые элементами навигации, расположенными в элементе управления *Toolbar*.

Таким образом, загружаем *ClassWizard,* переходим на закладку *Message Maps,* отыскиваем группу идентификаторов, отвечающих за навигацию по записям источника данных(ID_RECORD_FIRST, ID_RECORD_LAST, ID_RECORD_NEXT, ID_RECORD_PREV), и создаём обработчики.

Рассмотрим подробно создание обработчика для идентификатора ID_RECORD_ FIRST. Для этого после выбора мышью в окне *Object IDs* идентификатора ID_RECORD_ FIRST в окне *Messages* выберем *COMMAND.* После этого нажмём кнопку *“Add Functions”*. В результате этого появится диалог с предложением создать метод с именем *OnRecordFirst* (рис. 1.8).

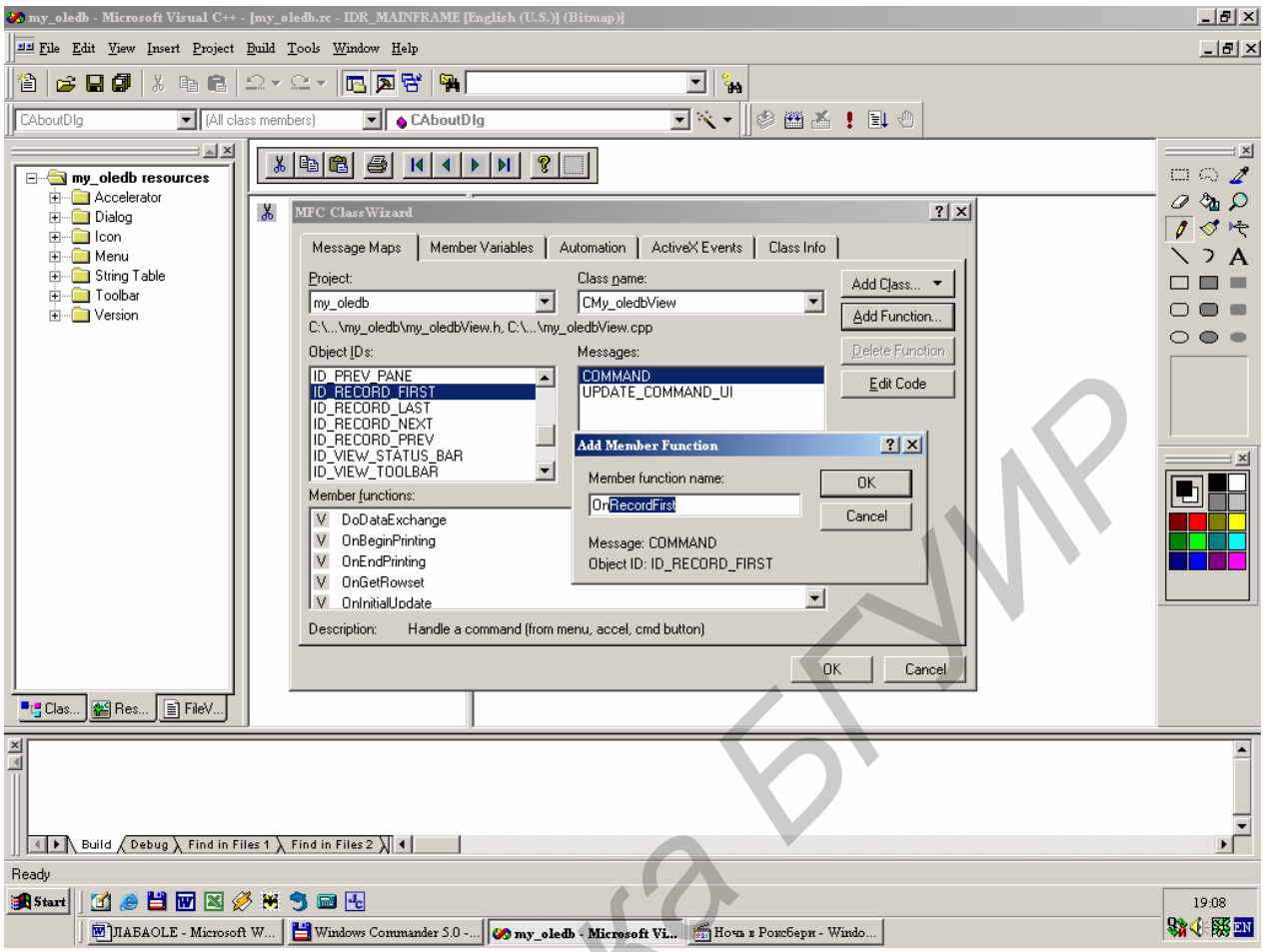

Рис. 1.8. Диалог определения имени метода OnRecordFirst

Оставляем предлагаемое название, для чего нажмём кнопку *"Ok"*. После этого нажмём кнопку *"Edit Code "* и перейдём к редактированию метода. Таким образом, в тело метода *OnRecordFirst()* нам необходимо добавить строки

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

Функция *MoveFirst()* перемещает курсор источника данных на первую позицию (т.е. на первую запись), а следующий за ней вызов метода *displayMembersOfDataSourse()* приводит к отображению данных в элементах EditBox.

В результате добавления обработчиков на все кнопки навигации мы должны получить следующий код в классе *CMy_oledbView*:

```
void COledbView::OnRecordNext() {
m_pSet->MoveNext();//переход на следующую позицию источника данных
          displayMembersOfDataSourse();}
```

```
void COledbView::OnRecordPrev() {

m_pSet->MovePrev();//переход на предыдущию позицию источника данных
                    displayMembersOfDataSourse();}

void COledbView::OnRecordLast() {
m_pSet->MoveLast();//переход на последнюю позицию источника данных
                    displayMembersOfDataSourse();}
void COledbView::OnRecordFirst()
{

m_pSet->MoveFirst();//переход на первую позицию источника данных
                     displayMembersOfDataSourse();
}
```

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

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

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

```
void COledbView::FillOleDBParameters(){
CString string;
wchar_t str[26];//объявление массива типа wchar_t

m_edit_name.GetWindowText(string);
memset(str, 0, 26); //выделение необходимой памяти

for(int i=0;i<string.GetLength();i++)
str[i] = (unsigned char)(LPCTSTR)string[i];      //инициализация массива str
                    m_edit_id.GetWindowText(string);
//инициализация переменной m_id_table
m_pSet->m_id_table = atoi((LPCTSTR)string);

// инициализация переменной m_name_table
wcscpy(m_pSet->m_name_table,str);

//обновление блока выделенной памяти
memcpy(m_pSet->m_name_table, &str[0],26);
}
```

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

Добавление данных

Для добавления данных в источник необходимо создать обработчик нажатия мышью на кнопку *"Insert"*. Таким образом, после того как мы дважды нажмем мышью на кнопку с названием *"Insert",* появляется диалог *"Add member function*", в котором нам предлагается создать метод обработки кнопки *"Insert"* с названием *"OnButtonInsert"*. Согласимся с предлагаемым названием и нажмём кнопку *"Ok"*. После этого в тело метода *OnButtonInsert()* добавим код :

```
void COledbView::OnButtonInsert()
{
FillOleDBParameters();
//добавление новых значений переменных в источник данных
m_pSet->Insert();
}
```

Поскольку метод *FillOleDBParameters()* уже определен, то можно произвести вставку данных в источник. Для этого вызываем метод *Insert(),* он выполнит все необходимые действия по добавлению значений в источник данных.

Редактирование данных

Для редактирования данных в источнике необходимо создать обработчик нажатия мышью на кнопку *"Update"*. Таким образом, дважды нажав мышью на кнопку с названием *"Update",* вызываем диалог *"Add member function",* в котором нам предлагается создать метод обработки кнопки *"Update"* с названием *"OnButtonUpdate"*. Согласимся с предлагаемым названием и нажмём кнопку *"Ok"*. После этого в тело метода *OnButtonUpdate()* добавим код :

```
FillOleDBParameters();
  m_pSet->SetData();//изменение значений полей источника данных
 // передача источнику данных информации об изменениях,
 //выполненных над строкой.
 m_pSet->Update(NULL,NULL);
```

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

Удаление записи

Для удаления записи необходимо создать обработчик нажатия мыши на кнопку *“Delete”*. Таким образом, дважды нажав мышью на кнопку с названием *“Delete”*, вызываем диалог *“Add member function”,* в котором нам предлагается создать метод обработки кнопки *“Delete”* с названием *“OnButtonDelete”*. Согласимся с предлагаемым названием и нажмём кнопку *“Ok”*. После этого в тело метода *OnButtonDelete()* добавим код :

```
void COledbView::OnButtonRemove()
{
m_pSet->Delete();
}
```

Для удаления записи вызывается метод *Delete()*.

Лабораторная работа №2

Построение приложения с использованием компонентов ActiveX

Цель работы:

1. Узнать назначение и возможности использования компонентов ActiveX.
2. Ознакомиться с процессом создания простого приложения доступа к источнику данных с использованием компонентов ActiveX.
3. Научиться использовать компоненты ActiveX для организации доступа к источникам данных.

2.1. Управляющие элементы ActiveX

Управляющий элемент ActiveX — независимый программный компонент, выполняющий специфические задачи стандартным способом. Разработчики могут задействовать один или несколько таких элементов в приложении, чтобы получить преимущества функциональных возможностей существующего программного обеспечения. В результате программное обеспечение построено в основном из уже готовых частей — это называется компонентным программным обеспечением.

Первоначально управляющие элементы ActiveX были известны как управляющие элементы OLE или OCX. Microsoft изменила название, чтобы отразить некоторые новые возможности, сделавшие эти элементы более подходящими для Интернета и WWW. Например, управляющий элемент ActiveX может хранить свои данные на странице где-то в WWW либо может быть выкачан с сервера WWW и затем запущен на машине клиента. И контейнер, в котором работает управляющий элемент, не обязан быть средой программирования — вместо этого он может быть средством просмотра WWW.

Управляющие элементы ActiveX — не отдельные приложения. Напротив, они являются серверами, которые подключаются к контейнеру элементов. Как обычно, взаимодействие между управляющим элементом и его контейнером определяется различными интерфейсами, поддерживаемыми COM-объектами. Фактически управляющие элементы ActiveX используют многие другие технологии OLE и ActiveX. Например, управляющие элементы обычно поддерживают интерфейсы для внедрения и зачастую предоставляют доступ к своим методам через диспинтерфейсы автоматизации.

Компоненты хранятся по категориям. Например, OLE-элементы управления хранятся в категории Registered ActiveX Control (OLE-Controls). В качестве компонентов выступают классы (возможно вместе с необходимыми ресурсами), элементы управления ActiveX (OCX), а также сложные компоненты. Компоненты можно включать в создаваемое приложение и использовать по своему усмотрению.

Количество страниц галереи компонентов и набор компонентов зависит от версии Visual C++ и постоянно расширяется. При работе с галереей можно создать собственную категорию, переименовать ее, добавлять или перемещать компоненты из одной категории в другую. В галерею компонентов можно включать компоненты, разработанные другими фирмами, а также собственные компоненты, разработанные самим программистом (в простейшем случае в качестве таких компонентов могут выступать классы созданных приложений). Главное, что любой компонент можно вставить в собственное приложение.

При работе с галереей можно создать собственную категорию, переименовать ее, добавлять или перемещать компоненты из одной категории в другую. При удалении компонента из галереи файлы, содержащие его, не удаляются. Их удаление при необходимости проводится вручную.

Компонент можно добавить в галерею четырьмя способами:

1) импортировать;

2) автоматически добавить OLE-элемент управления при его регистрации. (Автоматическая регистрация происходит только при первом добавлении элемента управления. Если же его удалить, а потом снова зарегистрировать, то, хотя сообщение о регистрации появляются, элемент управления в галерею автоматически не добавляется и его приходится импортировать);

3). добавить класс в момент его создания средством ClassWizard;

4). запустить программу setup для поставляемых компонентов.

Основное назначение галереи компонентов в том, что она позволяет вставлять в новый проект хранящиеся в нем компоненты.

Рассмотрим общую схему добавления компонентов в проект, уделяя основное внимание вставке OLE-элемента управления:

- открыть проект, в который необходимо добавить компонент;
- войти в галерею, выбрать нужную категорию и компонент;
- при помощи кнопки “Insert” добавить компонент в проект.

В зависимости от типа компонента предварительно может появиться окно с дополнительной информацией. Для OLE-элемента управления будет выведено окно, где указаны классы, которые будут добавлены в проект для обеспечения взаимодействия проекта и добавляемого элемента управления

ADO (Объекты данных ActiveX).

Технология ADO предлагает разработчику удобный прикладной интерфейс для OLE DB. ADO удобна в обращении, так как предоставляет объекты Automation, скрывающие интерфейсы OLE DB, что позволяет программисту уделять основное внимание решаемым задачам, а не сложностям технологии OLE DB.

ADO Data Control – это графический элемент управления на базе технологии ActiveX с кнопками навигации по записям. Он предоставляет приложению удобный интерфейс для работы с базами данных и позволяет избежать дополнительного кодирования. В *ADO Data Control* механизм ADO применяется для оперативного создания соединений между поставщиками данных и связанных с данными элементами визуализации. Элементы визуализации, связанные с

данными, представляют собой ActiveX-элементы пользовательского интерфейса с двумя важными свойствами:

- наличием параметра *DataSourse,* в котором можно задать идентификатор элемента *ADO Data Control*;
- способностью отображать данные, выбранные связанным с ним элементом *ADO Data Control*.

Когда элементы управления связаны с ADO Data Control, при просмотре записей все поля отображаются и обновляются автоматически. Такое поведение реализовано в самих элементах, и для этого не требуется ни одной дополнительной строчки кода. Примерами ActiveX-элементов для работы с данными являются, например *Microsoft DataGrid, Microsoft DataList*. Кроме того, допускается самостоятельно создавать собственные элементы управления, а также приобретать их у других поставщиков программного обеспечения.

2.2. Методические указания

Пусть наша программа называется *lab_actx*. Как указано в приложении 1 (см. лабораторный практикум по курсу "Визуальные средства разработки приложений"), создаем заготовку программы.

После этого мы можем перейти к добавлению элементов ActiveX *Microsoft ADO Data Control* и *Microsoft DataGrid Control* на форму.

Для этого выполним такую последовательность действий.

Выберем в меню пункт *Project*. В этом пункте – меню *Add to project*, а в нём - *Components and Controls* (рис. 2.1).

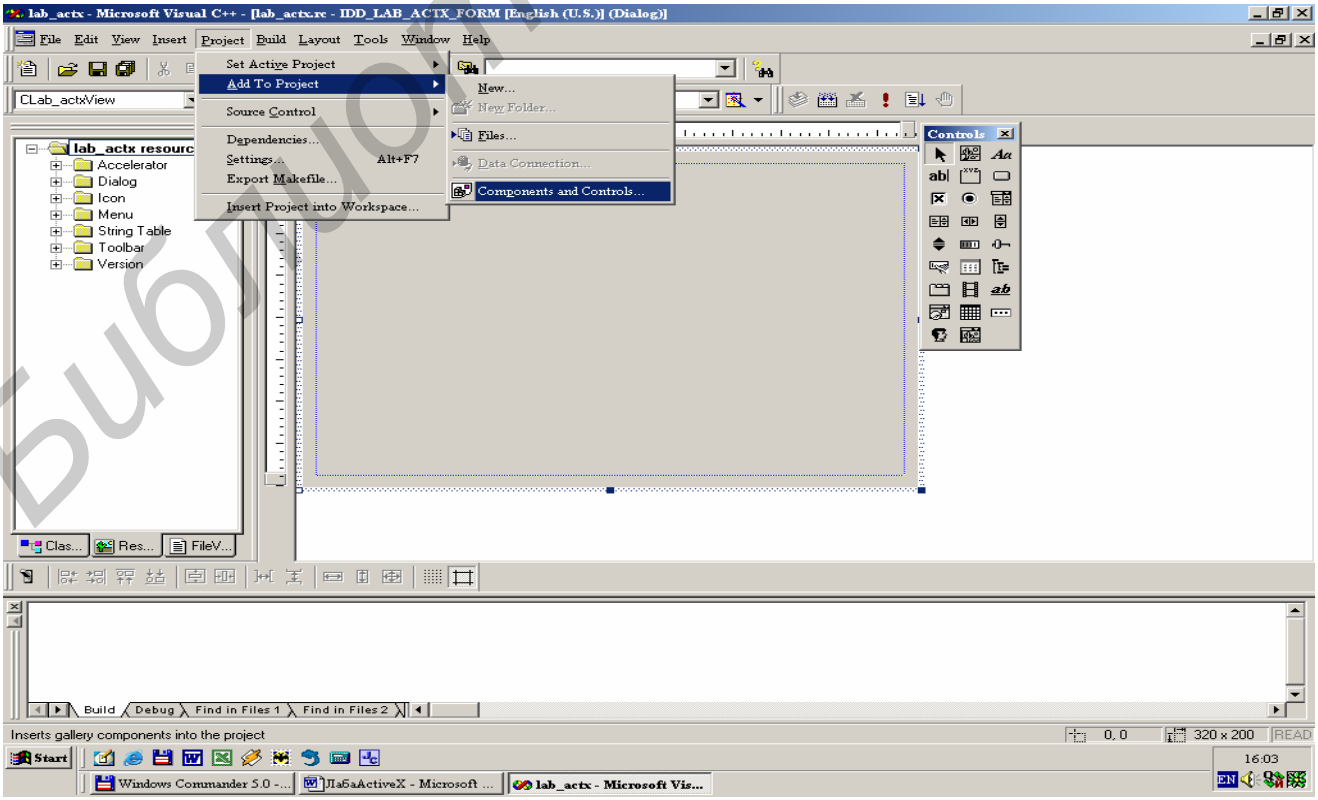

Рис.2.1. Выбор пункта меню Components and Controls

Щёлкнем этот пункт. В результате появится диалог *"Components and Controls Gallery"*- диалог выбора компонентов. Теперь в этом диалоге откроем папку *"Registered ActiveX Controls "*, в которой находятся все зарегистрированные в системе компоненты ActiveX. Теперь из списка элементов ActiveX выберем компонент *"Microsoft ADO Data Control, version 6.0 (OLEDB)"* и нажмём кнопку *"Insert"*. Сразу же мы увидим диалог подтверждения вставки выбранного компонента (рис. 2.2).

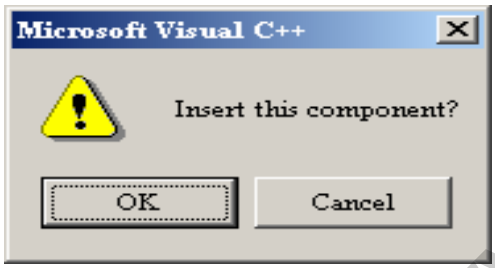

Рис.2.2. Диалог подтверждения вставки выбранного компонента

Согласимся и нажмём кнопку *"Ok"*. Теперь мы увидим новый диалог (рис.2.3), в котором нам предлагается подтвердить добавление классов, связанных с компонентом *"Microsoft ADO Data Control, version 6.0 (OLEDB)"*.

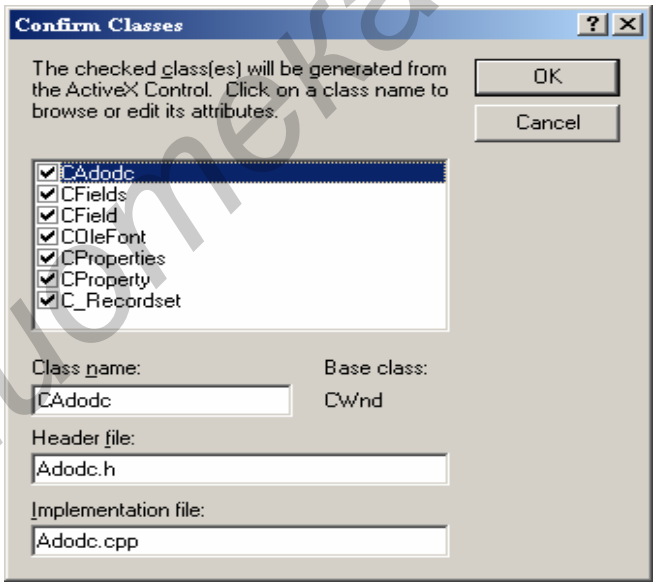

Рис. 2.3. Выбор классов, генерируемых для элемента ActiveX

Подтвердим его. После этого, классы, реализующие элемент управления *"Microsoft ADO Data Control*, будут добавлены в наш проект. Теперь таким же образом добавим компонент *"Microsoft DataGrid Control, Version 6.0 (OLEDB)"*. После этого панель инструментов должна выглядеть следующим образом (рис.2.4):

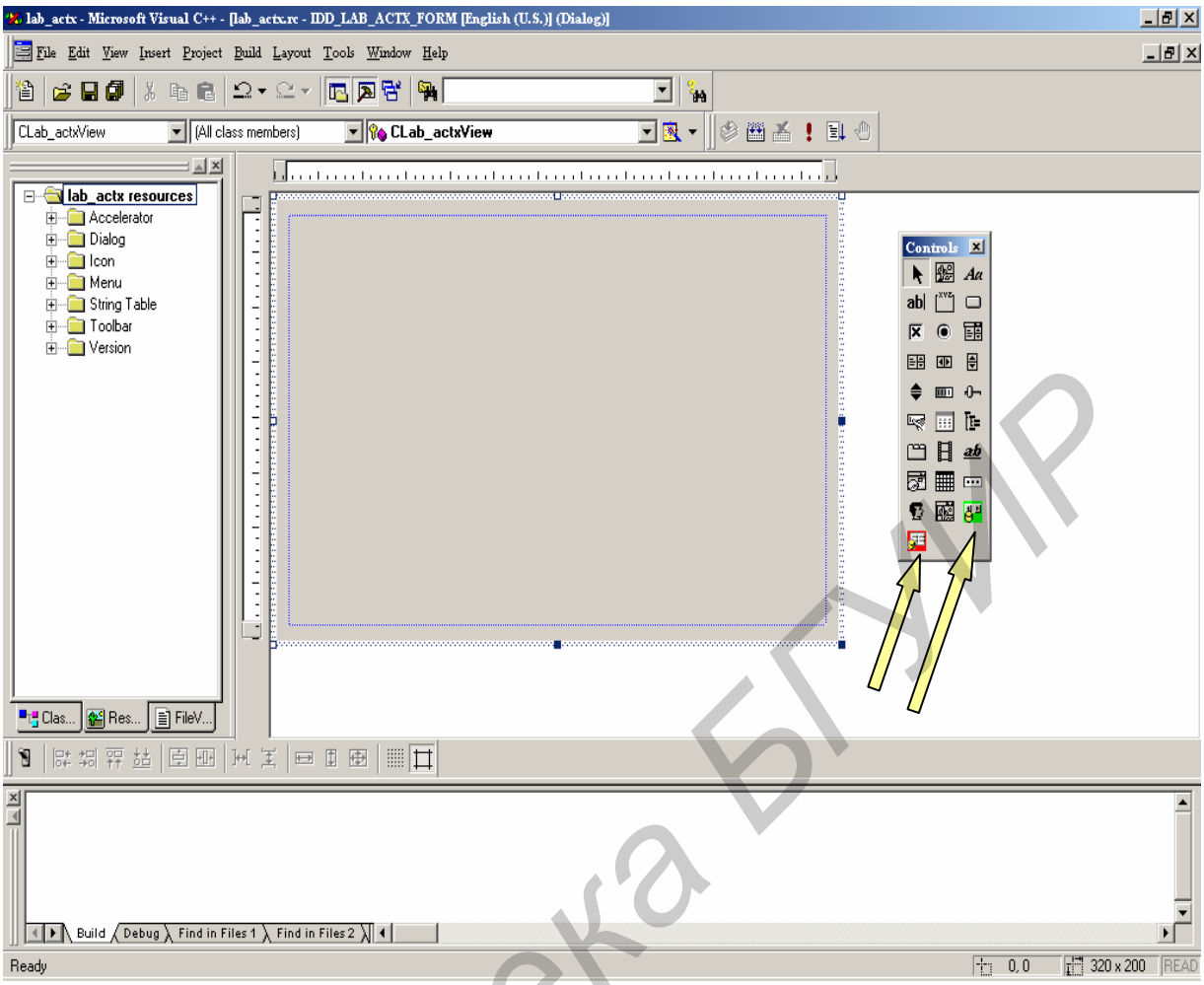

Рис.2.4. Панель инструментов после добавления элементов ActiveX

Для простоты последующего описания будем называть компонент *“Microsoft DataGrid Control, Versiongjc6.0 (OLEDB)”* – “Грид”, а компонент *“Microsoft ADO Data Control, version 6.0 (OLEDB)”* – “ Адо”.

Теперь добавим их на форму, после чего форма примет вид, показанный на рис.2.5.

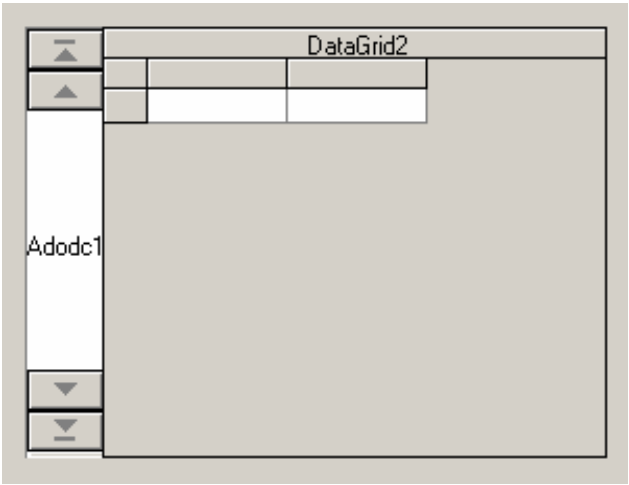

Рис.2.5. Вид формы после добавления на неё элементов ActiveX

Итак, начинаем настройку компонентов.

Прежде всего, займёмся компонентом Адо. Щёлкнем правой кнопкой мыши на этом компоненте. В появившемся контекстном меню выберем пункт *"Properties"*. Для изменения заголовка компонента выберем закладку *"General"* и в поле *"Caption"* напишем своё название (например *"Click"*). Теперь переходим на закладку *"Control"*, выбираем опцию *"Use ODBC Data Source Name",* после чего в ставшем активном элементе Combo Box выбираем нашу базу данных *Test*, как показано на рис. 2.6.

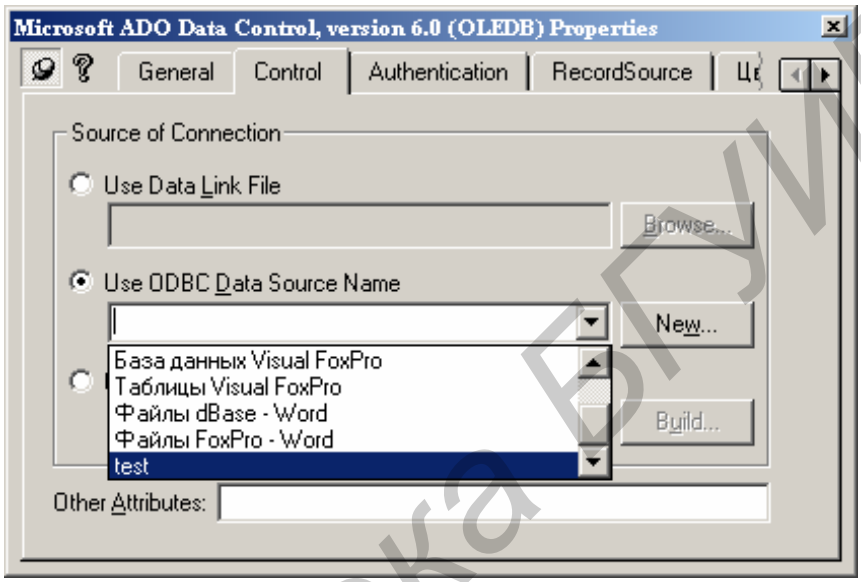

Рис.2.6. Выбор подсоединяемого источника данных

Теперь нам необходимо сформировать запрос к базе данных. Запрос к базе данных может быть представлен так:

- SQL-запрос к базе данных *(1-adCmdText);*
- подключение таблицы *(2-adCmdTable);*
- вызов хранимой процедуры из базы данных *(4-adCmdStoredProc).*

Выберем *"SQL-запрос к базе данных"*. Данный метод позволит нам осуществить выборку данных сразу из нескольких таблиц. Для этого перейдём в меню *"Properties"* элемента Адо и выберем закладку *"RecordSource"*. В элементе с именем *"Command Type"* выберем *"1-adCmdText"*, как показано на рис.2.7.

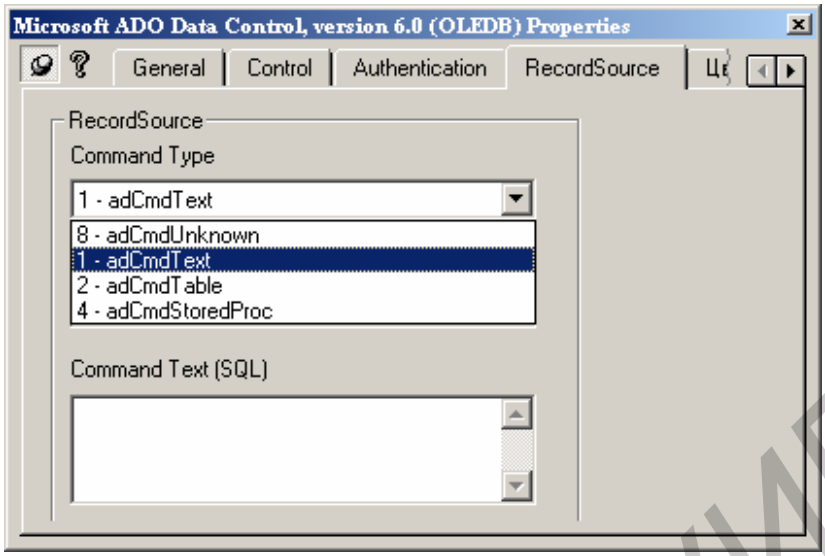

Рис. 2.7. Выбор метода извлечения информации из источника данных

Теперь в элемент с именем *"Command Text (SQL)"* введём SQL-запрос к нашей базе данных, который будет делать выборку сразу из двух таблиц:

SELECT [test_table].[id_table], [test_table].[name_table], [support_table].[String1], [support_table].[String2] FROM test_table INNER JOIN support_table ON [test_table].[id_table]=[support_table].[id_table];

Теперь наш проект готов к запуску.
При запуске на экране мы увидим следующее (рис 2.8).

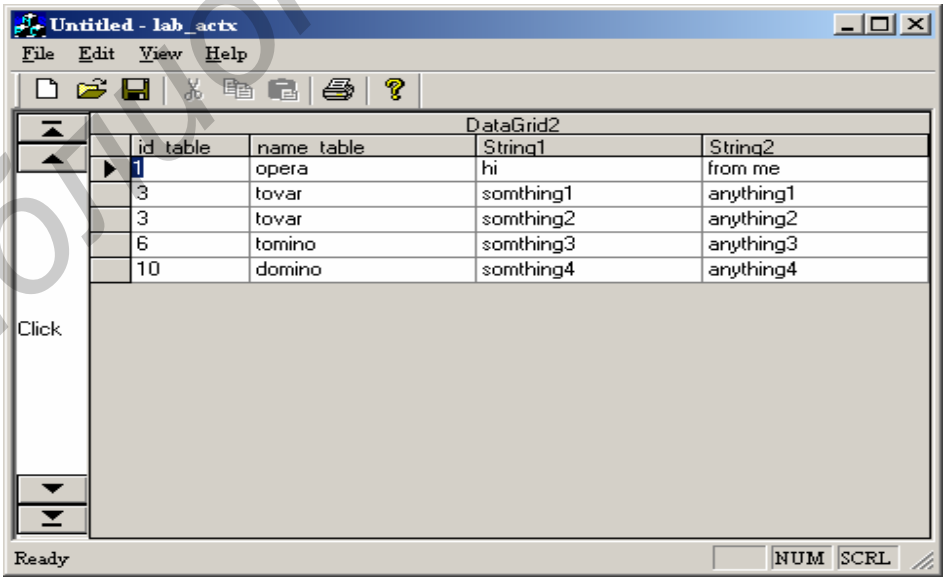

Рис.2.8. Внешний вид приложения

Навигация осуществляется по элементу Грид как с помощью элемента управления Адо, так и с помощью мыши.

Замечание. В текущий момент времени мы можем только лишь просматривать записи из источника данных.

Для того чтобы реализовать возможности удаления, добавления и редактирования, необходимо настроить свойства компонента Грид. Выполним такую последовательность действий.

Вызовем меню *"Properties"* элемента Грид. Перейдём на закладку *"Control"*. Поставим пометки напротив следующих свойств: *AllowAddNew, AllowDelete, AllowUpdate* (рис. 2.9).

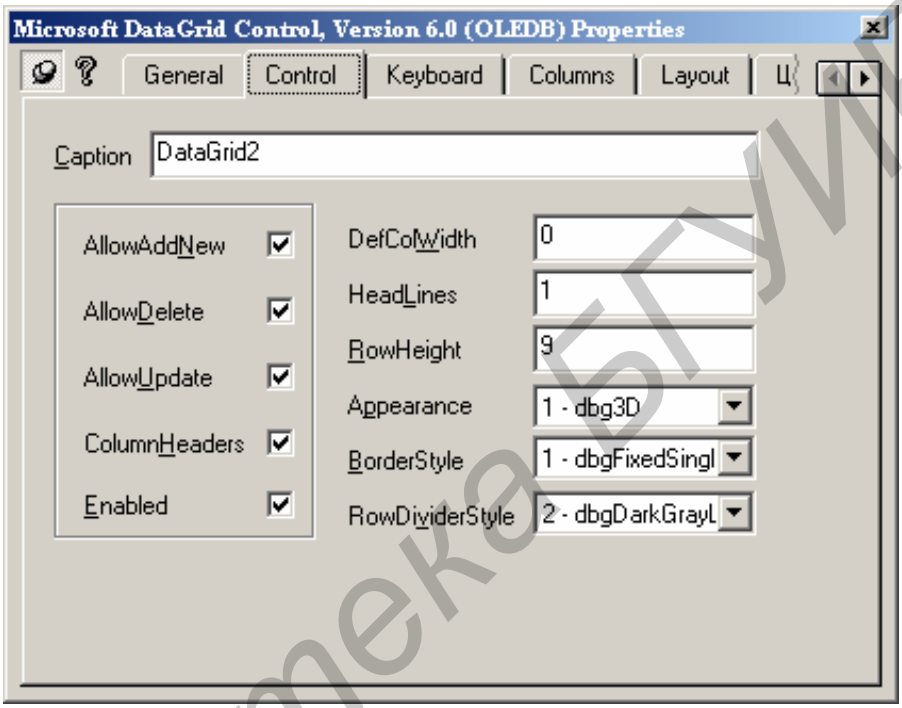

Рис. 2.9. Выбор свойств, разрешающих производить редактирование, удаление, добавление новых записей

Теперь перейдём к настройке свойств элемента Адо.

Для этого вызовем меню *"Properties"* элемента Адо, в котором выберем закладку *"All"*.

Теперь установим значения некоторых свойств:

"Cursor Locations" на *"2-Use server cursor"*

"Cursor Type" на *"1-Keyset Cursor Type"* (рис. 2.10, 2.11).

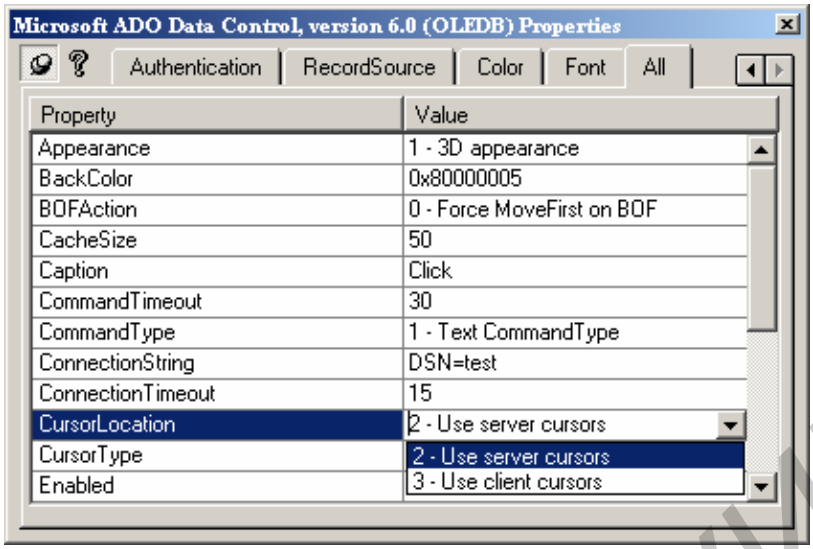

Рис. 2.10. Диалог изменения свойства “Cursor Locations”

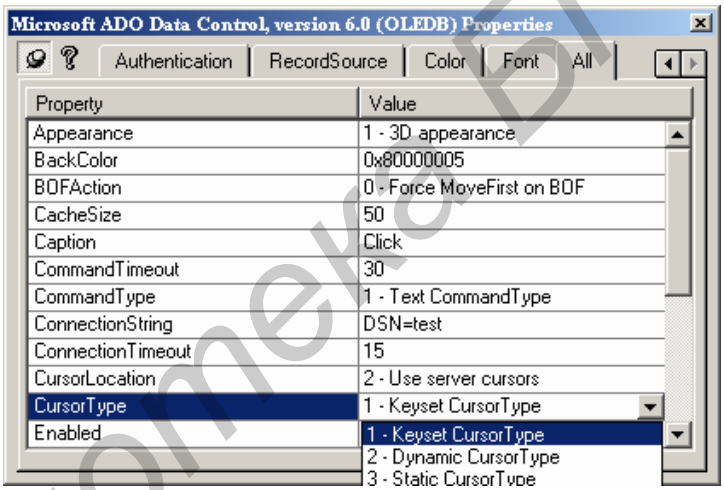

Рис.2.11. Диалог изменения свойства“Cursor Type”

Теперь, после запуска приложения, переместимся в ячейку, которую хотим редактировать.

Отредактируем её. Для подтверждения изменений, сделанных в ячейке, необходимо убрать фокус, например, нажатием клавиши *“Enter”*.

Для добавления новой записи необходимо переместиться в самую нижнюю строку элемента Грид и добавить данные. После завершения заполнения этой строки данными и перемещения фокуса на другую строку к элементу Грид автоматически добавляется ещё одна строка, которая располагается ниже текущей.

Удаление записей производится следующим образом. Для удаления нужной строки необходимо щелкнуть мышью по заголовку строки, таким образом выделив целую строку. После этого, при нажатии клавиши *“Delete”* на клавиатуре, происходит удаление выбранной строки (рис. 2.12).

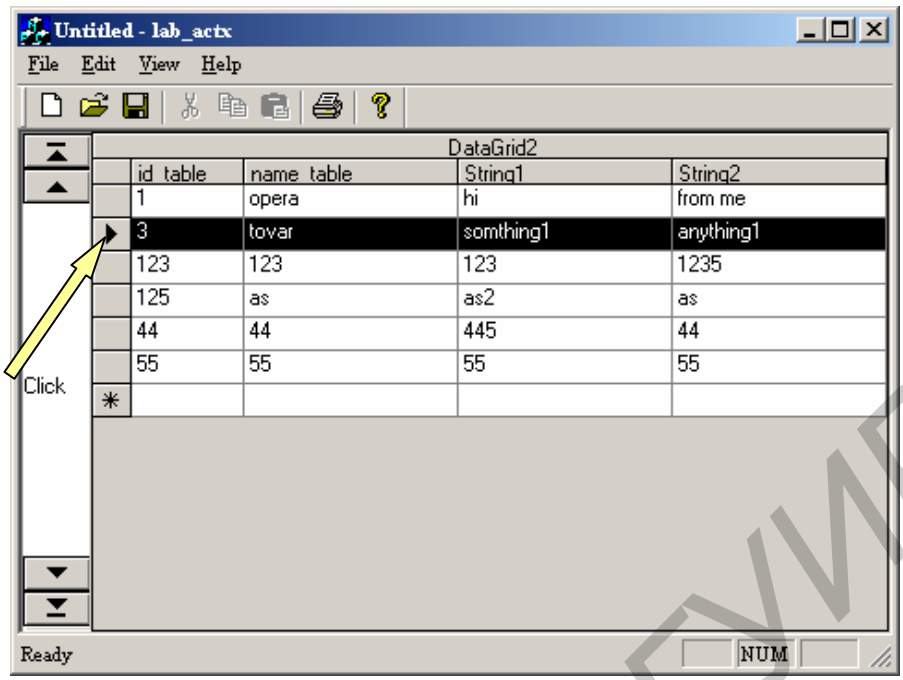

Рис. 2.12. Пример выделения удаляемой строки

Лабораторная работа №3

Использование потоков в приложении

Цель работы:

1. Ознакомиться с принципами организации многозадачности в Win32-приложениях.
2. Освоить программирование потоков в Win32-приложении;
3. Изучить способы синхронизации работы потоков и доступа к данным.
4. Создать приложение, демонстрирующее основные возможности многопоточности.

3.1.Потоковая многозадачность

Потоковая многозадачность - это одно из важных улучшений Windows, которое служит причиной существенного повышения производительности системы. В современных версиях Windows поддерживается два типа многозадачности.

Первый тип основан на процессах. Такая многозадачность поддерживалась уже с первых версий Windows. Процесс - это программа, или задача, которая выполняется. В многозадачных системах такого типа две и более программы могут выполняться одновременно.

Второй тип многозадачности основан на потоках. Такая многозадачность поддерживается оболочкой Win32 и используется в Windows 95 и Windows NT. Поток - это часть выполняющегося процесса. В Windows 95/NT каждый процесс имеет по крайней мере один поток, но потоков процесса может быть и два, и больше.

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

С введением потоковой многозадачности возникла необходимость в специальном механизме, называемом синхронизацией. Синхронизация позволяет контролировать выполнение потоков (и процессов) строго определенным образом. В Win32 для синхронизации выделена целая подсистема. Библиотека классов MFC полностью поддерживает средства многозадачности.

Использование потоков.

Потоковая многозадачность дает возможность программисту контролировать выполнение отдельных частей программы. Важно понимать, что все процессы имеют по крайней мере один поток выполнения. Он называется главным, первичным потоком. Но в пределах одного и того же процесса можно создавать несколько потоков. В общем случае, когда новый поток создается, он сразу же начинается выполняться. Таким образом, каждый процесс начинается с одного потока, к которому впоследствии могут добавляться дополнительные потоки.

Когда они создаются, родительский процесс начинает выполняться не последовательно, а параллельно.

3.2.Потоки MFC

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

В MFC потоковая многозадачность реализуется с помощью класса CWinThread. Кстати, производным от него является класс CWinApp, формирующий поток приложения.

При использовании классов, отвечающих за работу в многозадачном режиме, в программу следует включать стандартный библиотечный файл afxmt.h.

При создании многопотоковых программ наиболее часто используются именно рабочие потоки - необходимость в нескольких каналах сообщений возникает достаточно редко, однако во многих приложениях используются вспомогательные потоки, позволяющие вести фоновую обработку данных. Сосредоточимся поэтому на рабочих потоках (важно понимать, что на уровне API и рабочие, и интерфейсные потоки обрабатываются одинаково; различие между ними существует только в иерархии классов MFC).

Создание рабочего потока.

Для создания рабочего потока предназначена функция AfxBeginThreadбиблиотеки MFC:

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

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

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

Значение параметра pParam функции AfxBeginThread передается потоковой функции в качестве параметра. Это 32-разрядное число может использоваться для любых целей.

Начальный приоритет потока указывается в параметре nPriority. Если этот параметр равен 0, то используются установки приоритета текущего (родительского) потока.

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

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

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

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

3.3. 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. Разделяемым называется ресурс, доступ к которому могут одновременно получать несколько выполняющихся задач.

Механизм, обеспечивающий процесс синхронизации, называется ограничением доступа. Необходимость в нем возникает также в тех случаях, когда один поток ожидает события, генерируемого другим потоком. Естественно, должен существовать какой-то способ, с помощью которого первый поток будет приостановлен до совершения события. После этого поток должен продолжить свое выполнение.

Имеется два общих состояния, в которых может находиться задача. Во-первых, задача может выполняться (или быть готовой к выполнению, как только получит доступ к ресурсам процессора). Во-вторых, задача может быть блокирована. В этом случае ее выполнение приостановлено до тех пор, пока не освободится нужный ей ресурс или не произойдет определенное событие.

В Windows имеется специальные сервисы, которые позволяют определенным образом ограничить доступ к разделяемым ресурсам, ведь без помощи операционной системы отдельный процесс или поток не может сам определить, имеет ли он единоличный доступ к ресурсу. Операционная система Windows содержит процедуру, которая в течение одной непрерывной операции проверяет

и, если это возможно, устанавливает флаг доступа к ресурсу. На языке разработчиков операционной системы такая операция называется операцией проверки и установки. Флаги, используемые для обеспечения синхронизации и управления доступом к ресурсам, называются семафорами (semaphore). Интерфейс Win32 API обеспечивает поддержку семафоров и других объектов синхронизации. Библиотека MFC также включает поддержку данных объектов.

Объекты синхронизации и классы MFC.

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

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

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. Фактически это особая разновидность семафора.

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. Таким образом, данный объект сигнализирует о выполнении требуемого события.

При помощи объекта синхронизации четвертого типа можно запрещать выполнение определенных участков кода программы несколькими потоками одновременно. Для этого данные участки должны быть объявлены как критический раздел (critical section). Когда в этот раздел входит один поток, другим потокам запрещается делать то же самое до тех пор, пока первый поток не выйдет из данного раздела.

Критические разделы, в отличие от других типов объектов синхронизации, применяются только для синхронизации потоков внутри одного процесса. Другие же типы объектов могут быть использованы для синхронизации потоков внутри процесса или для синхронизации процессов.

В MFC механизм синхронизации, обеспечиваемый интерфейсом Win32, поддерживается с помощью следующих классов, порожденных от класса *CSyncObject:*

CCriticalSection - реализует критический раздел;

CEvent - реализует объект события;

CMutex - реализует исключающий семафор;

CSemaphore - реализует классический семафор.

Кроме этих классов в MFC определены также 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.

Когда какой-либо объект синхронизации создан, доступ к нему можно контролировать с помощью класса CSingleLock. Для этого необходимо сначала создать объект типа CSingleLock с помощью конструктора:

CSingleLock(CSyncObject pObject, BOOL bInitialLock = FALSE);*

Через первый параметр передается указатель на объект синхронизации, например семафор. Значение второго параметра определяет, должен ли конструктор попытаться получить доступ к данному объекту. Если этот параметр не равен нулю, то доступ будет получен, в противном случае попыток получить доступ не будет. Если доступ получен, то поток, создавший объект класса CSingleLock, будет остановлен до освобождения соответствующего объекта синхронизации методом Unlock класса CSingleLock.

Когда объект типа CSingleLock создан, доступ к объекту, на который указывал параметр pObject, может контролироваться с помощью двух функций: Lock и Unlock класса CSingleLock.

Метод Lock предназначен для получения доступа к объекту синхронизации. Метод Lock проверяет возможность доступа к ресурсу и приостанавливает поток, в случае занятости, до освобождения ресурса другими потоками, т.е. до тех пор, пока не будет получен доступ к ресурсу. Значение параметра определяет, как долго функция будет ожидать получения доступа к требуемому объекту. Каждый раз при успешном завершении метода значение счетчика, связанного с объектом синхронизации, уменьшается на единицу.

Метод Unlock освобождает объект синхронизации, давая возможность другим потокам использовать ресурс. В первом варианте метода значение счетчика, связанного с данным объектом, увеличивается на единицу. Во втором варианте первый параметр определяет, насколько это значение должно быть увеличено. Второй параметр указывает на переменную, в которую будет записано предыдущее значение счетчика.

При работе с классом CSingleLock общая процедура управления доступом к ресурсу такова:

1. Создать объект типа CSyncObj (например, семафор), который будет использоваться для управления доступом к ресурсу.
2. С помощью созданного объекта синхронизации создать объект типа CsingleLock.
3. Для получения доступа к ресурсу вызвать метод Lock.
4. Выполнить обращение к ресурсу.
5. Вызвать метод Unlock, чтобы освободить ресурс.

3.4. Методические указания

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

Использование потоков.

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

Интерфейсные и рабочие потоки MFC.

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

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

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

3.4.1. Создание рабочего потока

Для создания рабочего потока предназначена функция AfxBeginThread библиотеки MFC:

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

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

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

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

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

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

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

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

Данный указатель необходимо сохранять, если впоследствии предполагается обращение из родительского потока к созданному потоку (например, для изменения приоритета или для временного приостановления потока).

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

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

Для создания приложения Example выполните следующие действия:

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

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

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

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

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

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

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

Окончательный фрагмент кода в файле ExampleView.cpp представлен ниже.

```
UINT MyThread1(LPVOID pParam); // объявление функции потока 1
UINT MyThread2(LPVOID pParam); // объявление функции потока 2

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
UINT MyThread1(LPVOID pParam){
// через параметр передается указатель на окно просмотра
      CExampleView *ptrView=(CExampleView *)pParam;

      for(int i=0; i<100; i++)
      {
      CDC *dc=ptrView->GetDC();// получить контекст отображения
      Sleep(650); // Задержка на 650 миллисекунд
      CRect r;
      ptrView->GetClientRect(&r); //получить клиентскую область
                                  //окна
dc->TextOut(rand()%r.Width(),rand()%r.Height(),"*",1); // вывод
      }
      return 0;}

// определение функции потока 2
UINT MyThread2(LPVOID pParam){
for(int i=0; i<50; i++){
```

```
        Sleep(2000); // Задержка на 2000 миллисекунд
        AfxMessageBox("MyThread2"); // Вывод сообщения
        MessageBeep(0); // Подача звукового сигнала
}
return 0;
}
```

Откомпилируйте и запустите приложение. Не забудьте при компиляции установить в Project/Settings опцию многопотокового приложения, как это показано на рис. 3.1.

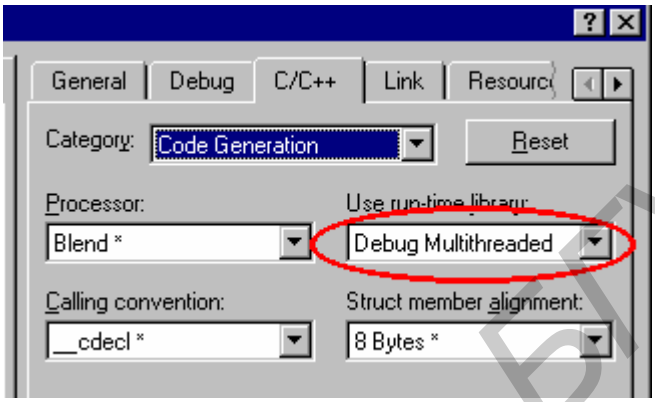

Рис 3.1.Пример выбора многопотокового приложения

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

Вообще говоря, поток выполняется до завершения своей потоковой функции. Поток может также “завершить сам себя” с помощью функции AfxEndThread библиотеки MFC. Параметр этого метода содержит статус завершения потока. Как правило, лучше давать потоку возможность нормально завершиться одновременно с потоковой функцией.

3.4.2.Остановка и возобновление выполнения потоков

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

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

3.4.3. Управление приоритетами потоков

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

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

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

Ниже показаны константы, соответствующие классам приоритетов в порядке убывания (по умолчанию программе присваивается приоритет *NORMAL_PRIORITY_CLASS*; причин менять его, как правило, нет):

REALTIME_PRIORITY_CLASS
HIGH_PRIORITY_CLASS
NORMAL_PRIORITY_CLASS
IDLE_PRIORITY_CLASS

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

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

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

THREAD_PRIORITY_TIME_CRITICAL
THREAD_PRIORITY_HIGHEST
THREAD_PRIORITY_ABOVE_NORMAL
THREAD_PRIORITY_NORMAL
THREAD_PRIORITY_BELOW_NORMAL
THREAD_PRIORITY_LOWEST
THREAD_PRIORITY_IDLE

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

3.4.4.Синхронизация потоков

Использование в программе нескольких потоков одновременно может привести к возникновению ряда специфических проблем. Например, как предотвратить одновременный доступ двух потоков к одним и тем же данным? Что произойдет, если в тот момент, когда один поток еще не завершил процедуру обновления некоторых данных, другой поток предпринимает попытку эти данные считать? Почти наверняка данные, считанные вторым потоком, окажутся некорректными, поскольку лишь некоторая их часть была на данный момент обновлена.

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

Объекты синхронизации и классы MFC.

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

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

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

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

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

одного процесса. Другие же типы объектов могут быть использованы для синхронизации потоков внутри процесса или для синхронизации процессов.

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

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

Для синхронизации работы потоков используются следующие классы:

CCriticalSection - реализует критическую секцию;

CEvent - реализует объект события;

CMutex - реализует исключающий семафор;

CSemaphore - реализует классический семафор.

Для контроля доступа используются следующие классы: CSingleLock и CMultiLock. Они контролируют доступ к объекту синхронизации и содержат методы, используемые для предоставления и освобождения таких объектов. Класс CSingleLock управляет доступом к одному объекту синхронизации, а класс CMultiLock - к нескольким объектам.

Когда какой-либо объект синхронизации создан, доступ к нему можно контролировать с помощью класса CSingleLock. Для этого необходимо сначала создать объект типа CSingleLock с помощью конструктора:

```
CSingleLock( CSyncObject* pObject, BOOL bInitialLock = FALSE );
```

Через первый параметр передается указатель на объект синхронизации, например семафор. Значение второго параметра определяет, должен ли конструктор попытаться получить доступ к данному объекту. Если этот параметр не равен нулю, то доступ будет получен, в противном случае попыток получить доступ не будет. Если доступ получен, то поток, создавший объект класса CSingleLock, будет остановлен до освобождения соответствующего объекта синхронизации методом Unlock класса CSingleLock.

Когда объект типа CSingleLock создан, доступ к объекту, на который указывал параметр pObject, может контролироваться с помощью двух функций: Lock и Unlock класса CSingleLock.

Метод Lock() предназначен для получения доступа к объекту синхронизации. Если объект синхронизации в данный момент не захвачен другим потоком, функция Lock() передаст этот объект во владение данному потоку. Теперь поток может получить доступ к защищенным данным. Завершив обработку данных, поток должен вызвать метод Unlock(), который освобождает объект синхронизации, давая возможность другим потокам использовать ресурс.

При работе с классом CSingleLock общая процедура управления доступом к ресурсу такова:

- создать объект синхронизации (например, семафор), который будет использоваться для управления доступом к ресурсу;

- с помощью созданного объекта синхронизации создать объект типа CSingleLock;
- для получения доступа к ресурсу вызвать метод Lock();
- выполнить обращение к ресурсу;
- вызвать метод Unlock(), чтобы освободить ресурс.

Важно уметь определять тот класс, который нужен для работы. Если приложение должно ждать некоторого события перед получением доступа, то нам нужен CEvent. Если к объекту будут иметь доступ несколько потоков из одного приложения и нужны ограничения по количеству потоков, тогда нам нужен CSemaphore. Если к объекту будет иметь доступ только один поток и из одного приложения – то CCriticalSection. Если к объекту будет иметь доступ только один поток, но из разных приложений – то CMutex.

Рассмотрим, как создавать и использовать объекты синхронизации.

3.4.5.Работа с семафорами

Рассмотрим, как обеспечить синхронизацию потоков на основе семафоров. Прежде всего необходимо создать семафор путем объявления объекта типа *CSemaphore.* Конструктор этого класса имеет следующий вид:

```
CSemaphore(LONG lInitialCount=1,LONG lMaxCount=1,
           LPCTSTR pstrName=NULL,
           LPSECURITY_ATTRIBUTES lpsaAttributes=NULL);
```

Семафоры имеют счетчик, указывающий количество задач, которым в настоящее время предоставлен доступ к ресурсу. Если значение счетчика равно нулю, то последующий доступ к ресурсу запрещается до тех пор, пока одна из задач не освободит семафор. Начальное значение счетчика семафора указывается в первом параметре конструктора. Обычно начальное значение задается равным единице, чтобы хотя бы один поток мог получить семафор. Допустимое число потоков, которым будет разрешен одновременный доступ, указывается во втором параметре. Если это значение равно единице, то семафор будет исключающим.

Третий параметр конструктора указывает на строку, содержащую имя объекта семафора. Поименованные семафоры становятся системными объектами и могут использоваться другими процессами. Когда два процесса вызывают семафоры с одинаковыми именами, обоим процессам будет предоставлен один и тот же семафор - это позволяет синхронизировать процессы. Вместо имени строки можно указать NULL - в этом случае семафор будет локализован внутри одного процесса. Последний параметр конструктора является указателем на набор атрибутов прав доступа, связанный с семафором. Если этот параметр равен NULL, то семафор наследует данный набор у вызвавшего его потока.

Модифицируйте приложение Example, добавив в него функции, использующие семафор. Для этого добавьте в меню Thread пункт Semaphore.

Функция OnSemaphore(), реализующая этот пункт, создает три потока, которые используют один и тот же ресурс. Одновременно доступ к ресурсу могут получить только два потока. Третий должен ждать, когда ресурс освободится.

Создавая семафор, вы передаете ему начальное и максимальное значения счетчика, как показано ниже:

```
CSemaphore Semaphore(2, 2);
```

Поскольку в этом примере семафоры будут использоваться для создания потокового класса, логично будет объявить указатель на объект класса CSemaphore в качестве переменной – члена потокового класса, а затем динамически создать объект класса CSemaphore в конструкторе потокового класса, как показано ниже:

```
semaphore = new CSemaphore(2, 2);
```

Теперь, когда объект семафора создан, можно начинать отсчет количества обращений к ресурсу. Для реализации процесса подсчета, прежде всего, необходимо создать экземпляр класса CSingleLock, передав ему указатель на семафор, который вы хотите использовать:

```
CSingleLock singleLock(semaphore);
```

Затем для уменьшения значения счетчика семафора вызывается метод Lock() класса CSingleLock:

```
singleLock.Lock();
```

На данный момент объект семафора выполнил уменьшение значения своего внутреннего счетчика. Это новое значение сохраняется до тех пор, пока объект семафора не будет освобожден посредством вызова его метода *Unlock():*

```
singleLock.Unlock();
```

Если сразу после освобождения семафора происходит выход объекта класса CSingleLock из области видимости (завершение функции, в которой он объявлен), метод Unlock() для объекта singleLock можно не вызывать. Деструктор объекта singleLock, вызванный при завершении работы функции, выполнит Unlock() автоматически.

Доступ к разделяемому ресурсу осуществим в классе CSomeResource. Класс имеет единственную переменную-член, являющуюся указателем на объект класса CSemaphore. Кроме того, в классе определены конструктор и деструктор, а также метод UseResource(), в котором непосредственно используется семафор.

Файл заголовка *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.

1. Добавьте в меню Thread пункт Semaphore и функцию OnSemaphore() в класс *CExampleView.*

2. Добавьте в проект два новых пустых файла SomeResource.h и SomeResource.cpp, пользуясь меню File->New. Выберите вкладку Files, типы файлов C/C++ Header File и C++ Source File.

3. Добавьте в эти пустые файлы тексты программ, приведенные выше.

4. Добавьте в файл ExampleView.cpp после директивы

```
#include "ExampleView.h"
```

директиву

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

5. Включите в начало файла, сразу же после директивы `#endif`, строку

CSomeResource someResource;

6. Добавьте в файл ExampleView.cpp перед функцией CExampleView::OnSemaphore() три следующие функции:

```
UINT ThreadProc1(LPVOID pParam)
{
  someResource.UseResource();
  AfxMessageBox("Thread1 had access.");
  return 0;
}

UINT ThreadProc2(LPVOID pParam)
{
  someResource.UseResource();
  AfxMessageBox("Thread2 had access.");
  return 0;
}

UINT ThreadProc3(LPVOID pParam)
{
  someResource.UseResource();
  AfxMessageBox("Thread3 had access.");
  return 0;
}
```

7. Добавьте в функцию *CExampleView::OnSemaphore()* следующие строки:

```
AfxBeginThread(ThreadProc1, this);
AfxBeginThread(ThreadProc2, this);
AfxBeginThread(ThreadProc3, this);
```

Теперь откомпилируйте новую версию приложения Example и запустите ее на выполнение. В раскрывшемся главном окне приложения выберите команду Threads->Semaphore. Приблизительно через 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. Третьему потоку потребовалось на пять секунд больше по той причине, что первые два потока первыми захватили контроль над ресурсом. Семафор в этой программе организован таким образом, что разрешает доступ к ресурсу только двум потокам одновременно. Таким образом, третий поток вынужден был ожидать, пока первый или второй поток освободит защищенный ресурс.

3.4.6.Работа с объектами событий

Объект события используется для оповещения процесса или потока о том, что произошло некоторое событие. Для работы с такими объектами предназначен класс CEvent. Конструктор класса имеет следующий прототип:

```
CEvent( BOOL bInitiallyOwn = FALSE, BOOL bManualReset =
FALSE,
      LPCTSTR lpszName = NULL,
      LPSECURITY_ATTRIBUTES lpsaAttribute = NULL );
```

Значение первого параметра определяет начальное состояние объекта. Если оно равно TRUE, то объект события установлен (событие произошло), а если FALSE, то объект не установлен или сброшен (событие не произошло).

Второй параметр указывает, каким образом состояние объекта будет изменяться при выполнении события. Если значение параметра равно TRUE (не ноль), то объект может быть сброшен только путем вызова метода ResetEvent класса CEvent. В противном случае объект автоматически сбрасывается после предоставления блокированному потоку доступа к ресурсу.

Третий параметр конструктора указывает на строку, содержащую имя объекта события. Поименованные объекты события становятся системными объектами и могут использоваться другими процессами. Вместо имени строки можно указать NULL - в этом случае объект события будет локализован внутри одного процесса.

Последний параметр конструктора является указателем на набор атрибутов прав доступа, связанный с объектом события. Если этот параметр равен NULL, то объект события наследует данный набор у вызвавшего его потока.

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 выполнение потока продолжится.

Чтобы продемонстрировать работу с объектами события, дополним наше приложение следующими функциями. Создадим два пункта меню: «Start Thread 2» и «End Thread 2». По пункту «Start Thread 2» должен запускаться процесс MyThread2, по пункту «End Thread 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.

Всем потокам процесса доступны все глобальные переменные процесса. Но может возникнуть ситуация, когда выполнение потока прерывается в тот момент, когда значение глобальной переменной является некорректным. Значение переменной может загружаться в регистр для выполнения некоторых операций. Если прерывание наступит в момент, когда значение переменной находится в регистре, а второй поток изменит ее значение в памяти, первый поток испортит ее значение, возвращая из регистра соответствующее значение. Одним из способов решения такой проблемы является объявление переменной как volatile, что гарантирует, что она не будет размещаться компилятором в регистре.

Выполните следующие действия для реализации объекта события и использования глобальной переменной.

1. С помощью редактора ресурсов добавьте команды Start Thread 2 и End Thread 2 в меню Thread приложения. Присвойте этим командам идентификаторы ID_STARTTHREAD2 и ID_ENDTHREAD2.

2. С помощью ClassWizard свяжите команду ID_STARTTHREAD2 с функцией обработки сообщения OnStartthread2(), команду ID_ENDTHREAD2 с функцией обработки сообщения OnEndhread2(). Перед тем как добавить новую функцию, убедитесь, что в поле Class Name выбрано значение CExampleView.

3. Добавьте в начало файла ExampleView.cpp приведенную ниже строку, поместив ее сразу после строки `#include "SomeResource.h"`:

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

Этот оператор подключает заголовочный файл для работы с классами объектов синхронизации.

4. Включите в начало файла ExampleView.cpp, сразу же после объявления

```
CSomeResource someResource
```

следующую строку:

```
volatile bool keeprunning;
```

5. Добавьте в начало файла ExampleView.cpp приведенные ниже строки, поместив их после строки `volatile bool keeprunning;`:

```
CEvent threadStart;
CEvent threadEnd;
```

6. Добавьте в функцию `UINT MyThread2(LPVOID pParam)` в файле ExampleView.cpp строки

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

перед циклом for. Первая функция создает объект `syncObjStart` класса `CSingleLock` для объекта события `threadStart`. Это вызов конструктора.

Вторая функция вызывает метод `Lock()` для этого объекта. Выполнение данного потока останавливается до тех пор, пока не произойдет событие для этого объекта.

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

```
MessageBeep(0);
```

строки

```
if (keeprunning==FALSE){
 AfxMessageBox("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.

8. Добавьте в функцию void CExampleView::OnStartthread2() строки:

```
 keeprunning=TRUE;//начальное значение переменной
                  keeprunning
threadStart.SetEvent();//объект события в состояние
                       «сигнализирует» (событие
                       произошло).
```

После ее выполнения метод Lock(), который ждет этого события, завершает свою работу, и выполняются следующие функции.

9. Добавьте в функцию void CExampleView::OnEndthread2() строку:

```
keeprunning=FALSE;
```

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

Использование критических секции

Критическая секция (Critical Section) – это участок кода, в котором поток получает доступ к ресурсу (переменной), который доступен из других потоков(рис. 3.2).

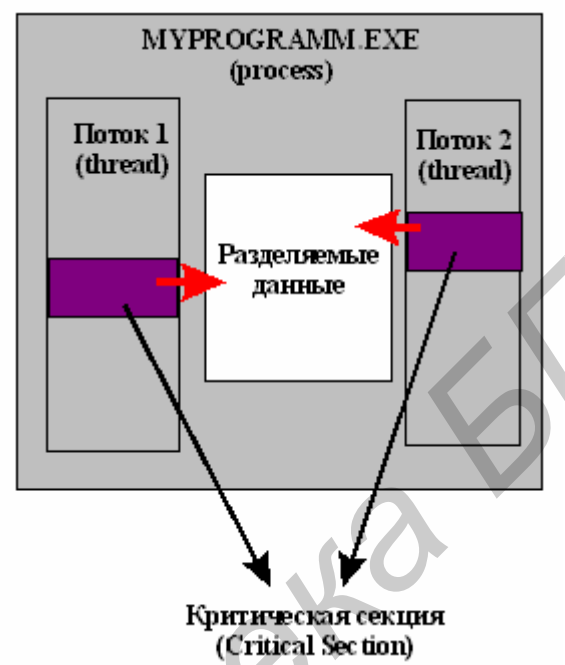

Рис. 3.2. Схема критической секции

Критические секции (разделы или секции кода, требующие монопольного доступа к разделяемым данным) удобны для управления доступом к данным.

Допустим, программа отслеживает показания времени как часы, минуты и секунды, а каждое из этих значений хранится в отдельной целочисленной переменной. Теперь представим, что значения времени совместно используются двумя потоками. Поток А изменяет значение времени и прерывается потоком Б после обновления часов, но до обновления минут и секунд. Результат: поток Б получает недостоверные показания времени.

Создавая критическую секцию, вы передаете потокам объект, который они должны использовать совместно. Любой поток, владеющий объектом критической секции, получает доступ к защищенным данным. Остальные потоки вынуждены ожидать освобождения критической секции, захваченной первым потоком, и только после этого какой-либо из них сможет захватить данную критическую секцию и в свою очередь получить доступ к данным. Доступ к защищенным данным может получить не более одного потока одновременно.

Для создания в программе, использующей библиотеку MFC, объекта критической секции необходимо создать экземпляр объекта класса `CCritical Section`, как это показано ниже:

```
CCriticalSection criticalSection;
```

Когда в программе необходимо получить доступ к данным, защищенным критической секцией, вызывается метод Lock() объекта этой критической секции, как показано ниже:

```
criticalSection.Lock();
```

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()` объекта критической секции:

```
criticalSection.Unlock();
```

Функция `Unlock()` освобождает объект критической секции. В результате другой поток сможет его захватить и получить доступ к защищенным данным.

Лучшим способом реализации механизма защиты данных является размещение критической секции в том же классе, где объявлены данные. Если это сделать, то основной программе не придется беспокоиться о синхронизации работы потоков — методы этого класса возьмут все на себя.

Для ознакомления с объектом критической секции создадим в разработанном приложении Example новый пункт Critical Section в меню Thread. 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.

Для реализации этого выполните следующие действия:

1. С помощью редактора ресурсов добавьте новый пункт Critical Section в меню Thread приложения. Присвойте этому пункту идентификатор ID_CRITICALSECTION.

2. С помощью ClassWizard свяжите команду ID_CRITICALSECTION с функцией обработки сообщения void CExampleView::OnCriticalsection(). Перед тем как добавить новую функцию, убедитесь, что в поле Class Name выбрано значение CExampleView.

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

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

В этом фрагменте текста программы последовательно вызываются функции `WriteThreadProc()` и `ReadThreadProc()`, каждая из них будет работать в своем собственном потоке.

4. Выполните необходимые действия по созданию класса `CCountArray`. Добавьте в проект два новых пустых файла `CountArray.h` и `CountArray.cpp`, пользуясь меню File->New. Выберите вкладку Files, типы файлов C/C++ Header File и C++ Source File.

Добавьте в пустой файл `CountArray.h` следующий текст.

```
#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 обыкновенный конструктор и деструктор, а также две функции для чтения и записи массива. Именно два последних метода класса и должны работать с объектом критической секции, так как только они имеют доступ к массиву.

5. 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 этого класса 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 номер один 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 номер два вызовет функцию SetArray() или GetArray(), то очередной вызов criticalSection.Lock() приостановит работу потока до тех пор, пока поток номер один 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 номер два, передав ему во владение объект критической секции.

6. Откройте файл CExampleView.cpp и добавьте в него приведенную ниже строку, поместив сразу после строки #include "afxmt.h":

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

7. Добавьте приведенную ниже строку в начало этого же файла, сразу после строки `volatile bool keeprunning;`

```
CCountArray countArray;
```

8. Добавьте в файл `ExampleView.cpp` перед функцией `void CExampleView::OnCriticalsection()` 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(str);
   }
 return 0;
}
```

Откомпилируйте новую версию приложения Example и запустите ее на выполнение. На экране раскроется главное окно приложения. Для запуска процесса выберите команду Thread->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 закрывать это окно сообщений. Первый поток записывает новые значения в массив ежесекундно, причем даже тогда, когда вы просматриваете содержимое массива с помощью второго потока.

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

Если вы внимательно проанализируете исходный текст программы, то увидите, что первый поток 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.

Второй поток 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. И наоборот, поток WriteThreadProc()

не сможет получить доступ к защищенным данным до тех пор, пока поток ReadThreadProc() не освободит объект критической секции.

Если вы хотите убедиться в том, что объект критической секции работает именно так, как описано выше, удалите строку criticalSection.Unlock(), расположенную в конце метода SetArray() класса CCountArray. Затем откомпилируйте и выполните программу. На этот раз после запуска потоков вы не увидите никаких сообщений. Поток WriteThreadProc() захватывает объект критической секции и не освобождает его, что заставляет систему остановить работу потока ReadThreadProc() раз и навсегда (или по крайней мере до окончания работы программы).

Использование исключающего семафора

Исключающие семафоры или мьютексы (mutex) во многом похожи на критические секции, но являются несколько более сложными объектами, так как обеспечивают безопасное разделение ресурсов не только потоками одного приложения, но также потоками различных приложений.

Рассмотрим текст файла заголовка CountArray2.h для класса CCountArray2. За исключением нового имени класса и объекта мьютекс, этот файл идентичен предыдущей версии файла CountArray.h.

```
#include "afxmt.h"

class CCountArray2
{
private:
    int array[10];
    CMutex mutex;
public:
    CCountArray2() {};
    ~CCountArray2() {};

    void SetArray(int value);
    void GetArray(int dstArray[10]);
}
```

Ниже приведен текст исходного файла CountArray2.cpp, реализующего этот модифицированный класс.

```
#include "stdafx.h"
#include "CountArray2.h"

void CCountArray2::SetArray(int value)
{
    CSingleLock singleLock(&mutex);
    singleLock.Lock();
```

```
    for (int x=0; x<10; ++x)
    array[x] = value;
}

void CCountArray2::GetArray(int dstArray[10])
{
    CSingleLock singleLock(&mutex);
    singleLock.Lock();

    for (int x=0; x<10; ++x)
    dstArray[x] = array[x];
}
```

Для получения доступа к объекту мьютекс необходимо создать объект класса CSingleLock для объекта mutex, как показано ниже:

```
CSingleLock singleLock(&mutex);
```

Аргумент конструктора является указателем на обеспечивающий синхронизацию потоков объект, с помощью которого и осуществляется управление. Затем для получения доступа к мьютексу вызывается метод Lock() объекта класса CSingleLock:

```
singleLock.Lock();
```

Если мьютекс еще не захвачен, то вызывающий поток становится его владельцем. Если владельцем мьютекса уже является другой поток, система приостанавливает работу вызывающего потока до тех пор, пока мьютекс не будет освобожден, после чего ожидающий поток сможет получить его в свое распоряжение и продолжить работу.

Для освобождения мьютекса необходимо вызвать метод Unlock() класса CSingleLock. Но поскольку вы создали экземпляр класса CSingleLock в стеке (а не в куче с помощью оператора new), то вызывать Unlock() вообще нет необходимости. Когда функция SetArray() завершит свою работу, объект выйдет из области видимости, что приведет к вызову его деструктора, который автоматически освободит объект.

Варианты индивидуального задания

1. Разработать приложение, которое может запускать один (поток Red), два (потоки Red и Green) или три потока (потоки 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 (потоки Red и Green) или три потока (потоки 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. Первый поток увеличивает значение счета на 1 в одну секунду. Второй поток 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 первым потоком, вторая часть– вторым. 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 первым потоком путем добавления единицы каждую секунду. Второй поток 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. По кнопке «Start» организуется запуск потока, который заполняет список некоторыми значениями.

8. Разработать приложение, которое выводит диалоговую панель с кнопкой «Array» и списком List box. По кнопке «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 исключающего семафора (Mutex).

Лабораторная работа №4

Программирование для Интернета с использованием Windows Sockets

Цель работы:

Научиться использовать протокол TCP/IP для программирования в Visual C++ (на основе создания двух приложений, общающихся посредством сокетов)

4.1. Сокеты, порты, адреса

Основным объектом, используемым в большинстве приложений для работы с сетью, является сокет. Сокеты были впервые использованы в системах UNIX в Калифорнийском университете в Беркли. Сокеты были изобретены для того, чтобы большинство сетевых соединений между разными приложениями могли быть осуществлены единообразно, так чтобы эти приложения могли работать с сокетами таким же образом, как эти приложения осуществляют чтение и запись файлов. С того времени сокеты претерпели значительные изменения, однако основа их осталась той же.

Во времена Windows 3.x, пока сетевые возможности не были встроены в операционную систему, пользователь мог приобрести различные сетевые протоколы от множества различных компаний. Каждая такая компания использовала протокол, который хотя бы немного отличался от протокола, использовавшегося другой компанией. В результате для каждого приложения требовалось иметь целый список различных сетевых приложений, с которыми это приложение могло работать. Многие разработчики приложений испытывали неудобства в связи с такой ситуацией. В результате основные компании, работающие с сетевыми технологиями, включая компанию Microsoft, собрались вместе и разработали стандарт Winsock (аббревиатура для Widows Socket) API. В результате появился стандартный интерфейс, позволяющий разработчикам приложений работать с сетью вне зависимости от конкретных используемых приложений.

Когда у нас возникает необходимость читать или запоминать файл, нужно обратиться к объекту, соответствующему этому файлу. Во многих создаваемых с помощью Visual C++ приложениях процесс обращения к файлам остается скрытым от нас. Сокет представляет собой объект, позволяющий осуществлять запись и считывание сообщений, которые будут пересылаться от одного приложения к другому.

Для того чтобы открыть сокет, нам необходимо знать, где расположен компьютер, на котором работает приложение, и номер порта, на котором это приложение ожидает вызов. Порт - это нечто, напоминающее дополнительный номер телефона, а адрес компьютера - это обычный телефонный номер. Если будет указан неправильный порт, то соединение может быть установлено не с тем приложением, с которым предполагалось, или, возможно, на запрос не будет отвечать ни одно приложение.

Только одно- единственное приложение может находиться в ожидании запроса на данном конкретном порте на компьютере. В то же время множество разных приложений, расположенных на одном и том же компьютере, могут ожидать запрос в один и тот же момент. Все эти приложения должны слушать на разных портах.

4.2. Модель клиент-сервер

Сервер представляет собой компьютер, хранящий данные, используемые другими компьютерами (клиентами). При этом запрос данных всегда исходит от клиента.

Связь клиент-сервер

1. При щелчке пользователя на гиперссылке в Web-броузере клиент определяет URL активизируемого соединения.
2. URL содержит информацию для клиента об адресе сервера, на котором должен находиться документ. Затем клиент обращается к серверу и организует с ним TCP/IP-соединение.
3. Клиент посылает запрос, выполнив преобразование информации из URL в формат, необходимый серверу. Кроме адреса сервера эта информация включает точное расположение запрашиваемого документа на сервере и желаемый протокол передачи.
4. Сервер пытается выполнить запрос и в случае успеха отсылает клиенту затребованные данные. В любом случае сервер посылает клиенту ответное сообщение.
5. Клиент получает данные и обрабатывает их.

Роль компьютера (сервер или клиент) определяется установленным на нем программным обеспечением. Для выполнения компьютером роли сервера необходимо:

- подключить компьютер к сети Интернет (для компьютера, к которому выполняется мало обращений, достаточно быстродействующего модема);
- задать IP-адрес, с помощью которого клиенты смогут обращаться к серверу;
- установить соответствующие программные средства (серверную часть) для обработки поступающих запросов.

Для выполнения компьютером роли клиента необходимо:

- подключить компьютер к сети Интернет;
- установить соответствующие программные средства (клиентскую часть) для формирования запросов данных, расположенных на других компьютерах (например Web-броузер).

Существуют приложения, состоящие из серверной и клиентской части. Каждый компьютер может выполнять функции как клиента, так и сервера, в зависимости от того, какая часть приложения на нем установлена.

Приложения могут использовать множество сетевых возможностей, и все эти возможности используют свойства интерфейса Winsock. Спецификацией Windows Sockets определяется интерфейс динамически загружаемой библиотеки, файл который, как правило, называется WINSOCK.DLL или WSOCK32.DLL.

Функции этой библиотеки реализуются разработчиками. Приложения могут вызывать эти функции и быть уверенными, что имя, смысл аргументов и поведение каждой функции не зависят от конкретной версии установленной библиотеки.

Поначалу программирование Winsock на Visual C++ сводилось к вызову библиотечных функций API. Многие производители разработали классы, в которых инкапсулированы вызовы этих функций.

4.3. Класс CAsyncSocket

Класс *CAsyncSocket* инкапсулирует асинхронные вызовы Winsock. Он содержит набор полезных функций, использующих Winsock API.

Методы класса *CasyncSocket*

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

4.4. Создание сетевого приложения

В качестве примера сетевого приложения создадим 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 этого можно проверить созданное приложение в работе. 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. Это позволит затем сравнить списки полученных и посланных сообщений и убедиться, что они совпадают.

4.4.1. Создание каркаса приложения

Создаем новый MFC-проект с помощью мастера приложений AppWisard и даем этому проекту имя, например, Sock. На первом шаге указывается, что приложение будет диалоговым, на втором шаге не забываем указать, что приложение будет использовать поддержку для Windows Socket. Далее выбираем кнопку Finish.

Внешний вид окна и начальные действия

После того как создан каркас приложения, приступаем 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 нам понадобится пара окон для редактирования текста, в которых мы будем указывать имя компьютера и номер порта, на котором будет слушать сервер. Нам нужна командная кнопка, которая будет заставлять сервер приступать к прослушиванию на сокете или же заставлять клиента устанавливать соединение с сервером, а также кнопка, которая позволит закрывать соединение. Кроме того, нам понадобится окно для редактирования текста, в которое мы будем вводить сообщение, предназначенное для передачи другому приложению, и кнопка, которая будет осуществлять такую передачу.

Наконец, нужно иметь пару окон для списка, в которые будут помещаться переданные и полученные сообщения. Расположим все эти элементы управления в соответствии с тем, как показано на рис.4.1, а свойства элементов управления зададим в соответствии с табл. 4.2.

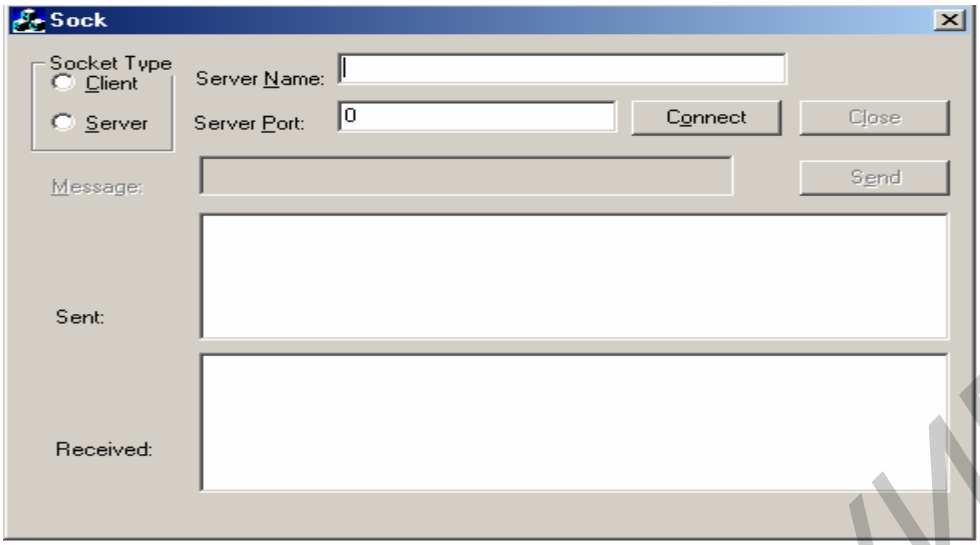

Рис.4.1. Расположение элементов управления

Таблица 4.2

Параметры элементов управления

Объект	Свойства	Значение
1	2	3
Group Box	ID	IDC_STATICTYPE
	Caption	Socket Type
RadioButton	ID	IDC_RCLIENT
	Caption	&Client
	Group	Checked
RadioButton	ID	IDC_RSERVER
	Caption	&Server
Static Text	ID	IDC_STATICNAME
	Caption	Server &Name:
Edit Box	ID	IDC_ESERVNAME
Static Text	ID	IDC_STATICPORT
	Caption	Server &Port:
Edit Box	ID	IDC_ESERVPORT
Command Button	ID	IDC_BCONNECT
	Caption	C&onnect
Command Button	ID	IDC_BCLOSE
	Caption	C&lose
	Disabled	Checked
Static Text	ID	IDC_STATICMSG
	Caption	&Message:
	Disabled	Checked
Edit Box	ID	IDC_EMSG
	Disabled	Checked
Command Button	ID	IDC_BSEND
	Caption	S&end
	Disabled	Checked
Static Text	ID	IDC_STATIC
	Caption	Sent:
List Box	ID	IDC_LSENT
	Tab Stop	Unchecket

Окончание табл. 4.2

1	2	3
	Selection	None
Static Text	ID	IDC_STATIC
	Caption	Received:
List Box	ID	IDC_LRECVD
	Tab Stop	Unchecket
	Sort	Unchecket
	Selection	None

После того как основа диалогового окна сконструирована, открываем «Матер классов» и создаем переменные для элементов контроля в соответствии с переменными элементов упраления.

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, используем имя функции OnRType. Вставим такую же функцию для события BN_CLICKED для элемента управления с идентификатором IDC_RSERVER. Функция имеет вид:

```
void CSockDlg::OnRType()
{
//Синхронизируем элементы управления в соответствии с переменными
UpdateDate(TRUE);
//В каком мы режиме?
if (m_iType == 0)  //Устанавливаем текст на кнопке
m_ctlConnect.SetWindowText("C&onnect");
else
m_ctlConnect.SetWindowText("&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.

4.4.2.Функции класса CAsyncSocket Class

Чтобы иметь возможность перехватывать и отвечать на события сокета, нам нужно 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 будем использовать для осуществления вызова функций каждого события сокета, предварительно осуществим проверку наличия ошибок. Чтобы создать этот класс, выберем в меню Insert | New Class. В диалоговом меню создания нового класса New Class оставим тип класса таким, какой предлагается по умолчанию (MFC), введем имя класса, например, CMySocket, и укажем в качестве базового класса *CAsyncSocket* в списке доступных базовых классов. Нажимаем кнопку ОК, новый класс вставлен.

После того как новый класс сокета создан, вставим переменную в класс, который будет использоваться в качестве указателя на родительское диалоговое окно. Указываем тип переменной CDialog*, имя переменной m_pWnd, доступ private. В классе необходимо определить метод, а значит, вставим новую функцию в этот класс. Тип функции void, объявим функцию в виде SetParent(CDialog* pWnd), доступ public. Редактируем созданную функцию.

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

В классе сокета создаем функции обработки событий. Для создания функции, соответствующей событию OnAccept, вставим новую функцию в класс сокета, тип функции void, опишем функцию в виде OnAccept(int nErrorCode), доступ protected. Редактируем код.

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

Вставляем подобные функции для событий OnConnect, OnClose, OnReceive и OnSend. После того как функции вставлены,нужно вставить заголовочный файл в диалоговое окно нашего приложения в класс сокета.

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

После того как требуемые функции событий созданы в классе сокета, вставим 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 - связана с другим приложением. Поскольку у нас существует два объекта сокета, то в диалоговый класс (CSockDlg) вводим две переменные. Обе переменные имеют тип класс сокета (CMySocket) и доступ private. Имя одной переменной m_sListenSocket, эта переменная связана с прослушиванием запроса на соединение, вторая переменная называется , m_sConnectSocket и используется для пересылки сообщения в том и в другом направлении.

После того как переменные сокета созданы, необходимо написать код, инициализирующий эти переменные. По умолчанию зададим тип приложения такой, как "клиент", номер порта 4000. Помимо этих параметров установим указатель в объектах сокетов, 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, которое обозначает компьютер, на котором мы работаем. Это внутреннее имя компьютера, превращаемое в адрес 127.0.0.1. Это имя и адрес компьютера, широко используемое в тех случаях, когда требуется осуществить соединение с другим приложением, установленным на том же самом компьютере, на котором мы и работаем.

```
BOOL  CSockDlg::OnInitDialog()
{
CDialog::OnInitDialog();
//...
SetIcon(m_hIcon, FALSE);
//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);
retutn TRUE;      //}
```

Связываемся с приложением

Когда пользователь нажимает кнопку, то все функции основного окна становятся недоступными. В этот момент пользователь не может менять параметры программы. Сейчас обращаемся к функции Create в соответствующей переменной сокета в зависимости от того, используется наше приложение в виде сервера или в виде клиента. Затем мы обращаемся либо к функции Connect, либо к функции Listen, этим мы инициализируем соединение с нашей стороны. Чтобы вставить описанные функции в приложение, откроем мастер классов Class Wizard

и вставим функцию для сообщения о событии BN_CLICKED в кнопке Connect(ID IDC_BCONNECT). Редактируем код функции.

```
void CSockDlg::OnBconnect()
{
//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.Creatr();
//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_sListenSockrt.Listen();
}
}
```

Затем, чтобы завершить приложение, вставляем функцию обработки событий сокета в диалоговый класс: OnAccept и OnConnect. Эти функции вызываются в нашем классе сокета. Эти функции 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, мы можем включить окно для ввода текста сообщений, которые будет передавать другому приложению.

Чтобы вставить такую функцию в приложение, добавляем новую функцию в диалоговый класс CSockDlg. Указываем тип функции void, описываем функцию как OnAccept, доступ public. Редактируем код функции.

```
void CSockDlg::OnAccept()
{
//принимаем запрос на соединение
m_sListenSocket.Accept(m_sConnectSocket);
```

//включаем элементы управления вводимого текста
GetDlgItem(IDC_EMSG)->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. Мы также включаем кнопку Close, с ее помощью соединение закрывается со стороны клиента (но не сервера). Чтобы добавить в приложение описанные функции, в диалоговый класс CSockDlg вставляем новую функцию, тип новой функции void, описываем функцию в виде OnConnsct, доступ к функции public.

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

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

Запустим две копии приложения. Зададим, чтобы одна из копий работала в режиме сервера, щелкнем кнопку Listen, чтобы перевести его в состояние ожидания запроса на соединение. Все элементы управления при этом будут находиться в отключенном состоянии. Второй экземпляр программы запустим в режиме клиента и нажмем 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.

Посылаем и получаем сообщение

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, нажав кнопку 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. Используем «Мастер классов» для вставки функции, которая будет выполняться после наступления события нажатия кнопки IDC_BSEND. Редактируем функцию.

void CSockDlg::OnBsend()
{

```
int iLen;
int iSent;
//Обновляем элементы управления в соответствии с переменными
UpdateData(TRUE);
//Есть сообщение для посылки?
if (m_strMessage !="")
{
//Получаем длину сообщения
iLen = m_strMessage.GetLength();
//Посылаем сообщение
iSent=
m_sConnectSocket.Send(LPCTSTR(m_strMessage), iLen};
//Смогли послать?
if (iSent == SOCKET_ERROR)
{
}
else
{
// Добавляем сообщение в список
m_ctlSent.AddString(m_strMessage);
// Обновляем переменные согласно элементам управления
UpdateData(FALSE);
 }}}
```

При срабатывании функция OnReceive, что происходит в момент, когда приходит сообщение, мы извлекаем это сообщение из сокета, используя функцию Receive. После того как сообщение извлечено, мы преобразуем его в тип String и добавляем его 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 strRecvd;
//Получаем сообщение
iRcvd = m_sConnectSocket.Receive(pBuf, iBufSize);
//Получили что-либо?
if (iRcvd == SOCKET_ERROR)
{
}
else
{
//Отрезаем конец сообщения
pBuf[iRcvd] = NULL;
//Копируем сообщение в CString
```

```
strRecvd = pBuf;
//добавляем сообщение в список полученных сообщений
m_ctlRecvd.AddString(strRecvd);
// обновляем переменные в соответствии с элементами управления
UpdateData(FALSE);
}
}
```

Сейчас можно скомпилировать и запустить две копии приложения, соединив их друг с другом. После того как соединение будет установлено, можно ввести сообщение в одном приложении и послать его другому приложению.

Завершение соединения

Чтобы закрыть соединение, пользователь клиента может щелкнуть кнопкой 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, отредактируем код функции OnClose, изменив код.

```
void CSockDlg::OnClose()
{
// закрываем сокет
m_sConnectSocket.Close();
// выключаем элементы управления, ответственные за посылку сообщений
GetDlgItem(IDC_EMSG)->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 влючаем 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 используем мастер классов и с его помощью вставим функцию, соответствующую событию нажатия кнопки Close (IDC_BCLOSE). Редактируем код функции.

```
void CSockDlg::OnBclose()
{
  // TODO: Add your control notification handler code here
// вызываем функцию OnClose
OnClose();
 }
```

Сейчас после компиляции и запуска двух копий приложения мы сможем осуществить соединение между клиентской и серверной версией приложения и пересылать между ними сообщения в обоих направлениях, а потом разорвать соединение из клиентского приложения, нажав на кнопку Close (рис.4.2)

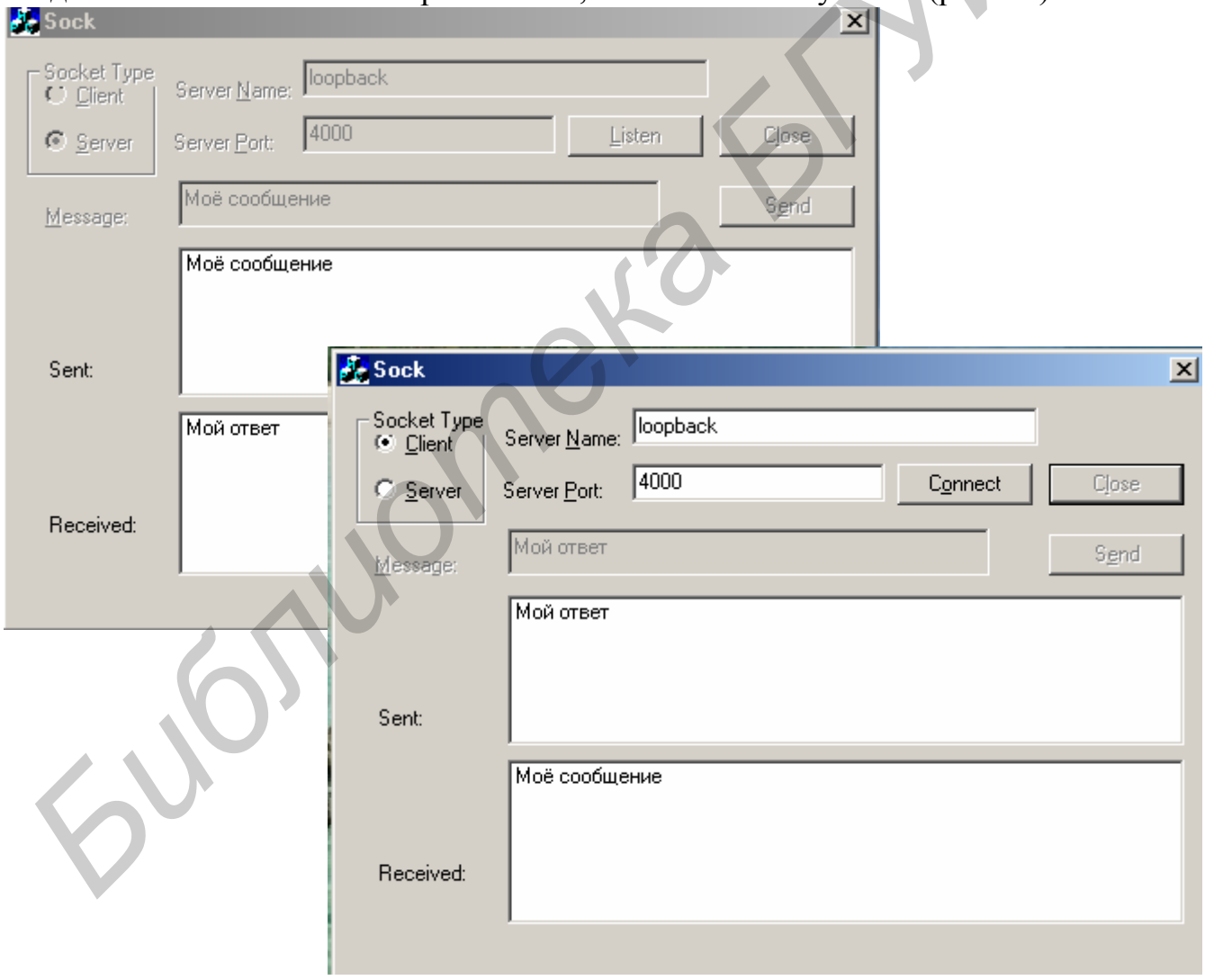

Рис 4.2.Результат работы приложения

Мы сможем восстановить соединение клиента с сервером, нажав на кнопку Connect еще раз. Если запустить третью копию приложения, изменим в ней номер порта, установим эту копию как сервер, включим режим ожидания запроса на

соединение, можно переключать клиентское приложение, поочередно подключаясь то к одному, то к другому, изменяя при этом номер порта.

Задание к лабораторной работе

Разработать приложение, позволяющее осуществлять взаимодействие клиента и сервера по совместному решению задач обработки информации.

Приложение должно располагать:

1) минимальным» пользовательским интерфейсом, определяющим возможности приложения;

2) возможностью передачи и модифицирования получаемых (передаваемых) данных;

3) средствами диагностики и обработки исключительных ситуаций, а также средствами поддержки деятельности пользователя.

Варианты индивидуального 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.

Литература

1. Грегори К. Использование Visual C++ 6.: Пер. с англ. - М.; СПб.; Киев.: Изд. дом «Вильямс», 2000. - 864 с.

2. Комличенко В.Н., Живицкая Е.Н., Соколов С.А. и др. Лабораторный практикум по курсу «Визуальные средства разработки приложений» для студентов специальности 40 01 02-02 "Информационные системы и технологии в экономике"./ -Мн.: БГУИР, 2002.- 89 с.

3. Тихомиров Ю. Самоучитель MFC. – СПб.: БХВ-Петербург, 2002. - 640 с.

Св. план 2003, поз. 111

Учебное издание

Комличенко Виталий Николаевич,
Едемская Оксана Павловна,
Кириенко Наталья Алексеевна и др.

ВИЗУАЛЬНЫЕ СРЕДСТВА РАЗРАБОТКИ ПРИЛОЖЕНИЙ

Учебно-методическое пособие
по курсу «Объектно-ориентированное проектирование и программирование»
для студентов специальности 40 01 02-02
«Информационные системы и технологии в экономике»
дневной формы обучения

Редактор Т.А.Лейко
Корректор Е.Н. Батурчик

Подписано в печать 27.01.2004.	Формат 60x84 1/16.	Бумага офсетная.
Печать ризографическая.	Гарнитура Times.	Усл. печ. л.4,07.
Уч.-изд. л. 4,0.	Тираж 150 экз.	Заказ 290.

Издатель и полиграфическое исполнение:
Учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники»
Лицензия ЛП № 156 от 30.12.2002.
Лицензия ЛП № 509 от 03.08.2001.
220027, Минск, ул. П. Бровки, 6