

## Тема 1. Архитектура Microsoft SQL Server.

### **Физическая архитектура базы данных**

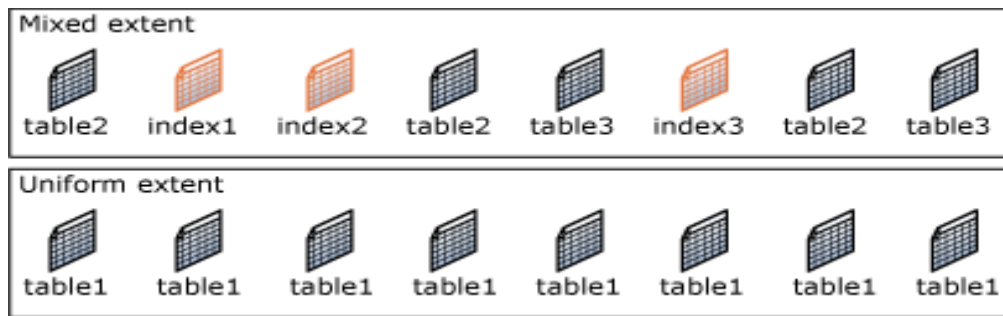
Экземпляр Microsoft SQL Server включает в себя системные базы данных (master.model, msdb, tempdb), содержащие служебную информацию, и пользовательские базы данных. Каждая база данных размещается в отдельных файлах – минимум двух: один для самой базы данных – файл данных (mdf-файл), и один для журнала транзакций (ldf-файл). Первый файл данных (mdf-файл) является основным и кроме самих данных содержит системную информацию, второй и все последующие файлы данных являются вторичными (ndf-файлами) и содержат непосредственно сами данные.

Расположение этих файлов можно указать при создании базы данных.

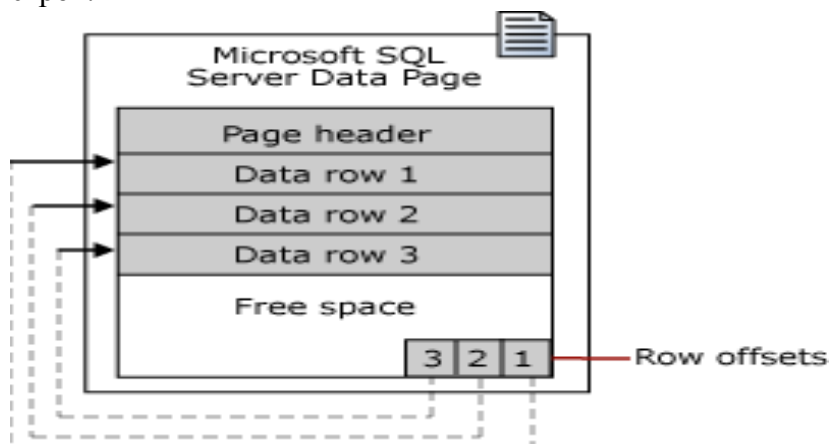
Основной единицей хранения данных является страница. SQL Server выполняет чтение и запись данных постранично. Вся база данных логически подразделена на страницы, нумеруемые начиная с 0. Размер страницы составляет 8 Кбайт (128 страниц на один мегабайт)

Для более эффективного управления страницами они объединяются в экстенды – по 8 страниц в экстенде. Экстенды могут быть двух типов:

- **mixed** – страницы, входящие в такой экстенд могут принадлежать разным объектам;
- **uniform**- экстенд содержит станицы, принадлежащие одному объекту.



Системная информация о странице хранится в заголовке, под который отводится первые 96 байт. Эта информация содержит номер страницы, тип страницы, количество свободного пространства, и ID объекта, владеющего страницей. В конце каждой страницы располагается таблица смещения строк.

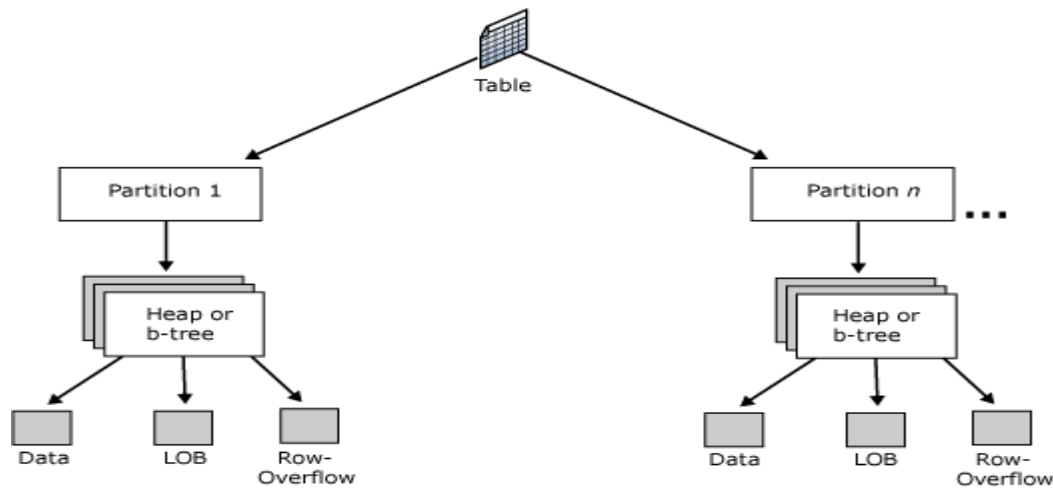


SQL Server использует в файлах данных следующие типы страниц:

- **Data** - страница содержит строки всех данных за исключением данных типа text, ntext, image, nvarchar(max), varchar(max), varbinary(max) и xml-данных;

- **Index** - страница содержит информацию о индексах;
- **Text/Image** страница для хранения:
  - следующих типов LOB-объектов:
    - text, ntext, image, nvarchar(max), varchar(max), varbinary(max) и xml-данных;
  - столбцов переменной длины, чей размер строки превышает 8 KB:
    - varchar, nvarchar, varbinary и sql\_variant
- **Global Allocation Map** - страница данного типа содержит информацию об используемости экстенгов (на одной странице хранятся данные об используемости 64000 экстенгов);
- **Shared Global Allocation Map** - страница данного типа содержит информацию об используемости экстенгов типа Mixed;
- **Page Free Space** - страница содержит информацию о количестве свободного пространства на странице;
- **Index Allocation Map** - страница содержит данные, какие экстенги имеют страницы, принадлежащие одному объекту-владельцу;
- **Bulk Changed Map** - страница содержит информацию об экстенгах, измененных посредством набора операций, выполненных после последней операции копирования базы данных (BACKUP LOG);
- **Differential Changed Map** - страница содержит информацию об экстенгах, измененных с момента последней операции копирования базы данных (BACKUP DATABASE).

Таблицы и индексы хранятся как наборы страниц. Таблица может быть подразделена на одно или несколько разбиений (partitions), содержащих строки.



Разбиение таблицы определяет пользователь при ее создании.

Сначала в базе данных создается функция, отображающая строки таблицы или индекса на разбиение, основанное на значениях указанного столбца (CREATE PARTITION FUNCTION).

Далее создается схема, которая отображает разбиения разбиваемых таблиц или индексов на группы файлов (CREATE PARTITION SCHEME) .

Например:

```
CREATE PARTITION FUNCTION myPFunc (int)
    AS RANGE LEFT FOR VALUES (1, 1000);
```

```
GO
```

```
CREATE PARTITION SCHEME myPScheme
    AS PARTITION myPFunc
```

```
TO ( to1fg, to1fg, to1fg, to2fg ); -- группы файлов
```

Имя группы файлов определяется командой CREATE DATABASE. По умолчанию файлы данных входят в основную группу файлов.

Таблицы SQL Server 2005 используют для организации их страниц данных в разбиении один из следующих двух методов:

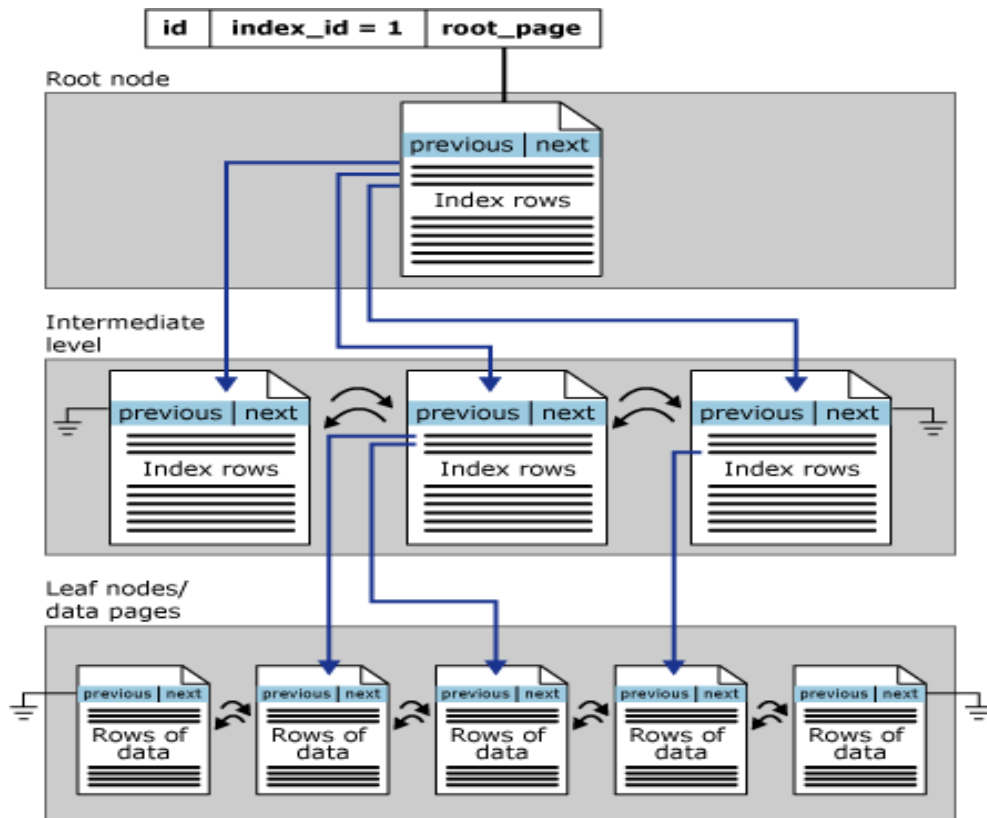
- **Кластерные таблицы** (для которых создан кластерный индекс, требующий физического перестроения данных в соответствии со структурой индекса);
- **Кучи** (Heaps) – таблицы, не имеющие кластерного индекса.

Индексы в SQL Server организованы в виде В-деревьев. Каждая страница в индексном В-дереве называется индексным узлом (index node). В вершине В-дерева расположен корневой узел (root node). В нижней части дерева располагаются индексные узлы, называемые листья (leaf nodes). Между вершиной дерева и листьями располагаются промежуточные уровни (intermediate levels). Каждая строка индекса содержит ключевое значение и указатель или на страницу промежуточного уровня в В-дереве, или на данные, расположенные в нижнем уровне дерева (leaf level).

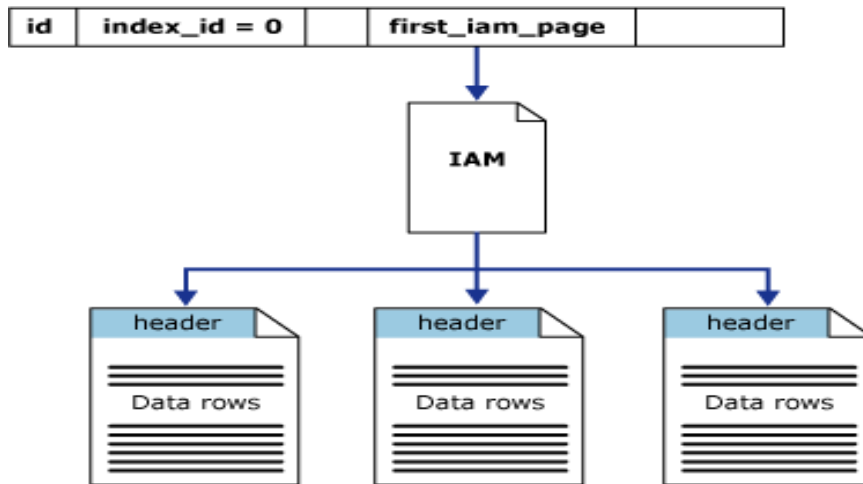
Если таблицы, для которой построен кластерный индекс имеет три разбиения, то для каждого разбиения будет построено свое В-дерево.

Страницы в цепочке данных и строки в них упорядочиваются согласно значению ключа кластерного индекса.

Следующий рисунок иллюстрирует структуру кластерного индекса.



Доступ к таблицам, не имеющим кластерных индексов, выполняется посредством последовательного просмотра IAM-страниц с целью нахождения экстентов, содержащих страницы, относящиеся к данной куче. Следующий рисунок иллюстрирует процесс извлечения строк данных таблицы машиной баз данных Database Engine.

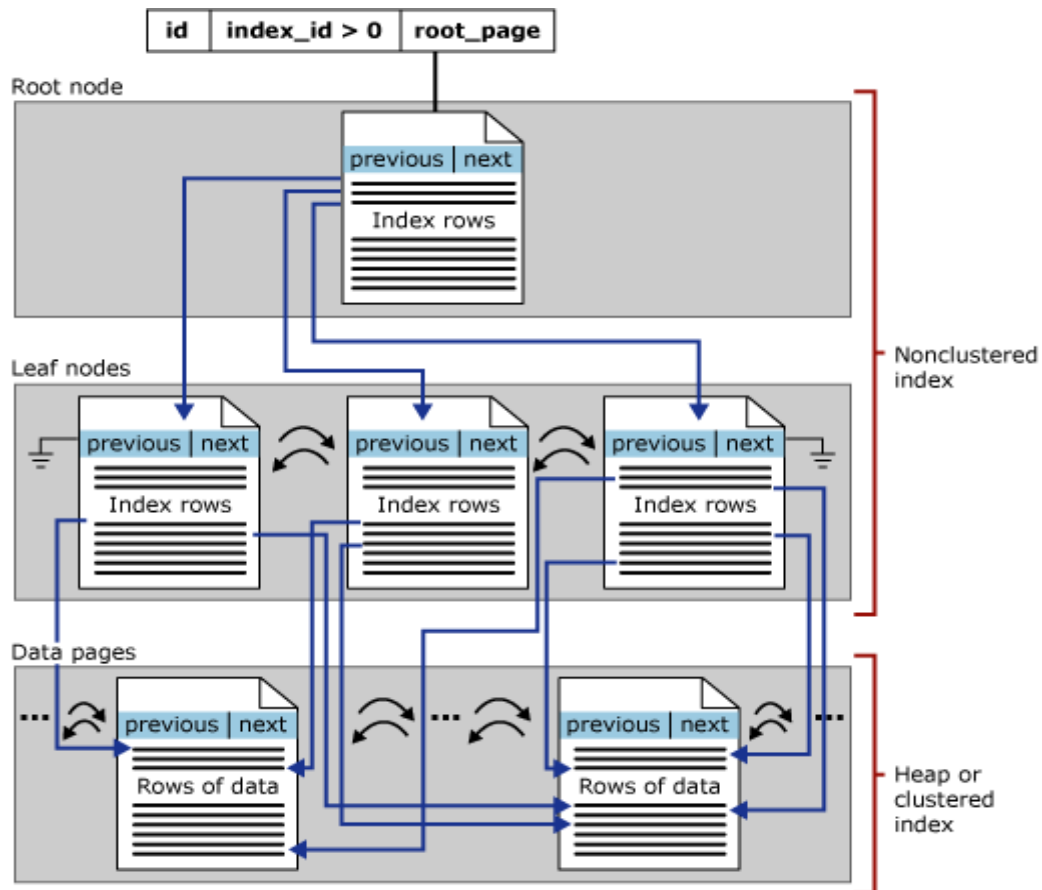


Некластерные индексы имеют структуру В-дерева подобно кластерным индексам, за исключением следующих различий:

- строки данных таблицы являются не упорядоченными и хранятся в порядке, основанном на их некластерном ключе;
- leaf- уровень некластерного индекса состоит из индексных страниц, а не страниц данных.

Каждая строка индекса в некластерном индексе содержит некластерное ключевое значение и локатор строки (row locator). Если таблица не содержит кластерного индекса, то локатор строки является указателем строки, в противном случае – ключом кластерного индекса для данной строки. Указатель строки (ROWID) содержит ID-файла, номер страницы, номер строки на странице.

На следующем рисунке представлена схема использования некластерного индекса.



Для столбцов типа XML SQL Server позволяет создавать XML-индексы.

Единица размещения (allocation unit) является набором страниц в куче или В-дереве, используемом для управления данными на основе типа страницы.

SQL Server для управления данными таблиц и индексов применяет следующие типы единиц размещения:

- IN\_ROW\_DATA
- LOB\_DATA



- ROW\_OVERFLOW\_DATA

### Схема

В версии Microsoft SQL Server 2005 схемой называется набор элементов (entities) базы данных, формирующий единое пространство имен. В предыдущих версиях Microsoft SQL Server понятие схемы было тесно связано с именем пользователя базы данных: для каждого пользователя предназначалась одноименная схема базы данных. В версии Microsoft SQL Server 2005 пользователь может размещать свои объекты в различных схемах.

Объединение объектов в схемы, не ассоциируемые с конкретным пользователем, не требует внесения изменений в процедуры, использующие объекты схемы, при замене пользователя.

Несколько пользователей могут владеть одной схемой через членство в роли или группе Windows.

Создать новую схему можно, используя графический инструментарий Microsoft SQL Server Management Studio, или программным путем, выполнив оператор CREATE SCHEMA, который имеет следующее формальное описание:

```
CREATE SCHEMA schema_def [<schema_element> [ , ...n ] ]

<schema_def> ::=
{
    ИМЯ_СХЕМЫ
  | AUTHORIZATION ИМЯ_ВЛАДЕЛЬЦА
  | ИМЯ_СХЕМЫ AUTHORIZATION ИМЯ_ВЛАДЕЛЬЦА
}

<schema_element> ::=
{
    определение_таблицы | определение_представления |
```

<code>grant_оператор   revoke_оператор   deny_оператор</code> <code>}</code>
---

Этот SQL-оператор одновременно с созданием новой схемы может добавлять в нее новые таблицы и представления, а также устанавливать полномочия данным объектам посредством фраз GRANT, DENY и REVOKE.

Имя владельца схемы является именем пользователя базы данных.

Определение таблицы задается оператором CREATE TABLE (при этом необходимо наличие соответствующих полномочий), а определение представления – оператором CREATE VIEW.

Например:

```
USE MyDb;
```

```
CREATE SCHEMA Schema1 AUTHORIZATION dbo
```

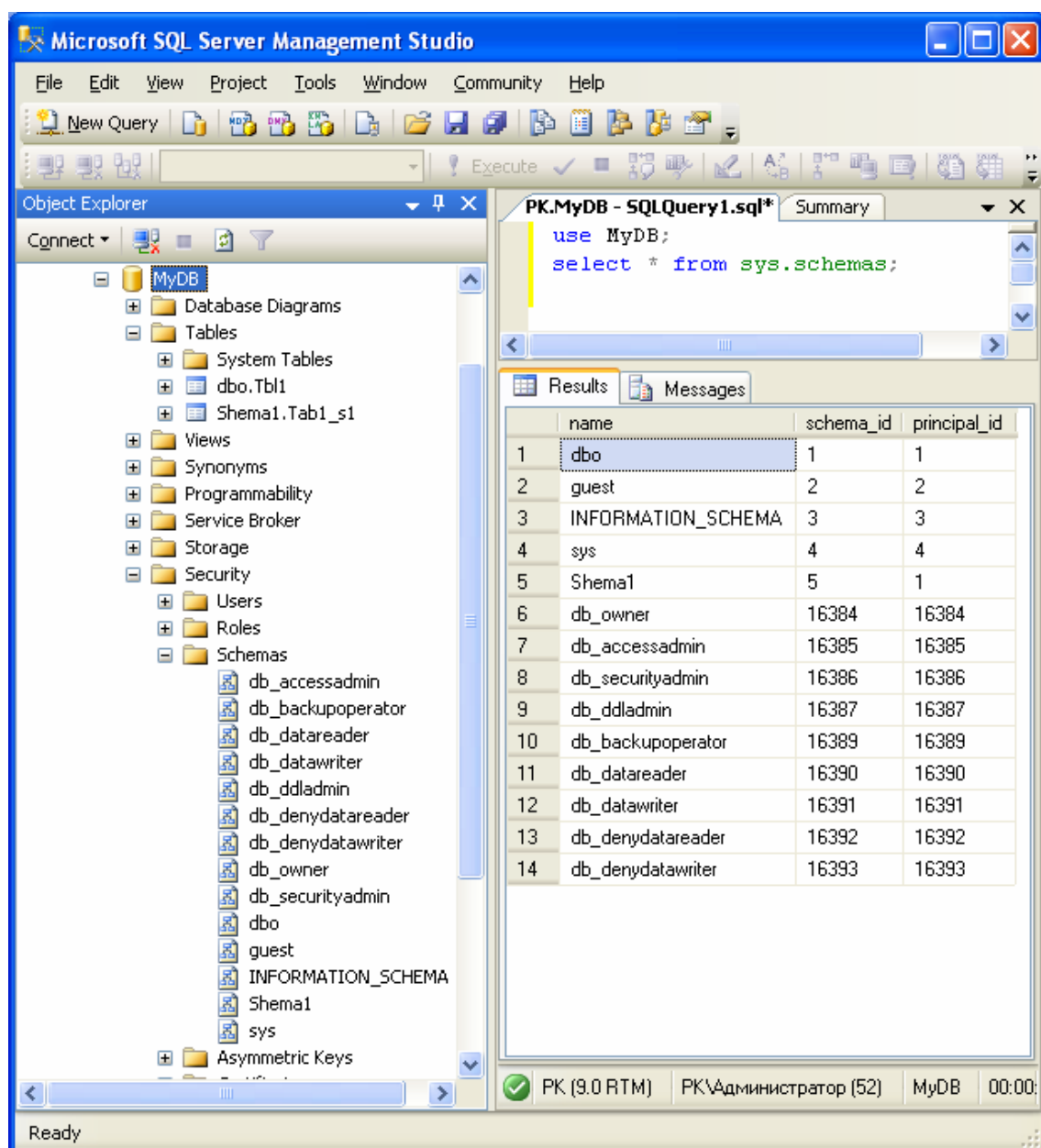
```
    CREATE TABLE Tbl1 (id1 int, f1 int, f2 varchar(20))
```

```
    GRANT SELECT TO user1
```

```
    DENY SELECT TO user2;
```

```
GO
```

Для получения списка всех форм можно использовать представление sys.schemas:



## **Объекты базы данных**

Логически данные в базе данных хранятся в виде объектов базы данных.

Объекты данных хранятся в схеме базы данных.

SQL Server предоставляет следующие объекты данных:

- таблицы;
- представления;
- синонимы;
- индексы;
- хранимые процедуры;
- триггеры;
- пользовательские типы данных;
- функции пользователя;
- ключи, обеспечивающие ссылочную целостность;
- ограничения целостности;
- умолчания
- правила (используются для обратной совместимости)

К объектам базы данных также относятся схемы, пользователи и роли.

В SQL Server введены новые объекты, используемые Service Broker:

- типы сообщений (структура сообщения, отправляемого от одного сервиса другому),
- контракты (соглашения между двумя сервисами),
- очереди (сообщения, направленные сервису),
- сервисы (наборы задач, где каждая задача представляется контрактом), сервисные программы.

## **Создание базы данных**

Создать базу данных можно как программным путем, выполнив SQL-оператор CREATE DATABASE, так и используя средства Microsoft SQL Server Management Studio.

При подключении к SQL-серверу следует выбрать тип сервера: Database Engine, Analysis Services, Reporting Services, SQL Server Mobile, Integration Services, и тип аутентификации пользователя: Windows Authentication или SQL Server Authentication.

Для создания новой базы данных следует в Microsoft SQL Server Management Studio в окне Object Explorer выполнить для секции Database команду контекстного меню New Database и в предложенном далее диалоге ввести имя создаваемой пользовательской базы данных.

На странице Options диалога New Database следует задать параметры базы данных.

Режим работы с курсором фиксируется двумя параметрами:

- **Close Cursor on Commit Enabled** – параметр определяет будет ли закрыт курсор при фиксации транзакции, в которой данный курсор был открыт. По умолчанию значение параметра равно false – курсор остается открытым после фиксации транзакции, и при откате транзакции закрываются курсоры, которые не были определены как STATIC или INSENSITIVE. Отметим, что при значении True фиксация или откат транзакции означает закрытие курсора;
- **Default Cursor** – параметр, определяющий по умолчанию поведение курсора, как LOCAL или GLOBAL (значение по умолчанию).

Для ограничения доступа к базе данных значение параметра Restrict Access можно задать как:

- **Multiple** – к базе данных одновременно разрешен доступ нескольким пользователям (значение по умолчанию);
- **Single** – в каждый момент времени к базе данных может быть подключен только один пользователь;
- **Restricted** – к базе данных могут быть подключены только пользователи, имеющие полномочия, определяемые ролями db\_owner, dbcreator или sysadmin.



## Тема 2. Реализация доступа к базам данных средствами языка SQL.

### Язык SQL

Язык SQL предназначен для доступа к информации и управления реляционной базой данных.

Язык SQL определяет:

- операторы языка, называемые иногда командами языка SQL;
- типы данных;
- набор встроенных функций.

По своему логическому назначению операторы языка SQL часто разбиваются на следующие группы:

- язык определения данных DDL (Data Definition Language);
- язык манипулирования данными DML (Data Manipulation Language).

Язык определения данных включает операторы, управляющие объектами базы данных. К объектам базы данных относятся таблицы, индексы, представления.

Язык манипулирования данными включает операторы, управляющие содержанием таблиц базы данных и извлекающими информацию из этих таблиц.

Язык DML определяет следующие операторы:

- SELECT – извлечение данных из одной или нескольких таблиц;
- INSERT – добавление строк в таблицу;
- DELETE – удаление строк из таблицы;
- UPDATE – изменение значений полей в таблице.

### Переключение между базами данных

Один пользователь может работать с несколькими базами данных. Для переключения на конкретную базу данных применяется SQL-оператор USE, который имеет следующее формальное описание:

<b>USE</b> { <i>имя_базы_данных</i> }
---------------------------------------

Все операторы, выполняющиеся после оператора USE будут использовать указанную базу данных.

Для выполнения перехода на другую базу данных пользователь должен обладать соответствующими полномочиями, например, иметь роль dbo или sysadmin.

### **Переключение между контекстами выполнения**

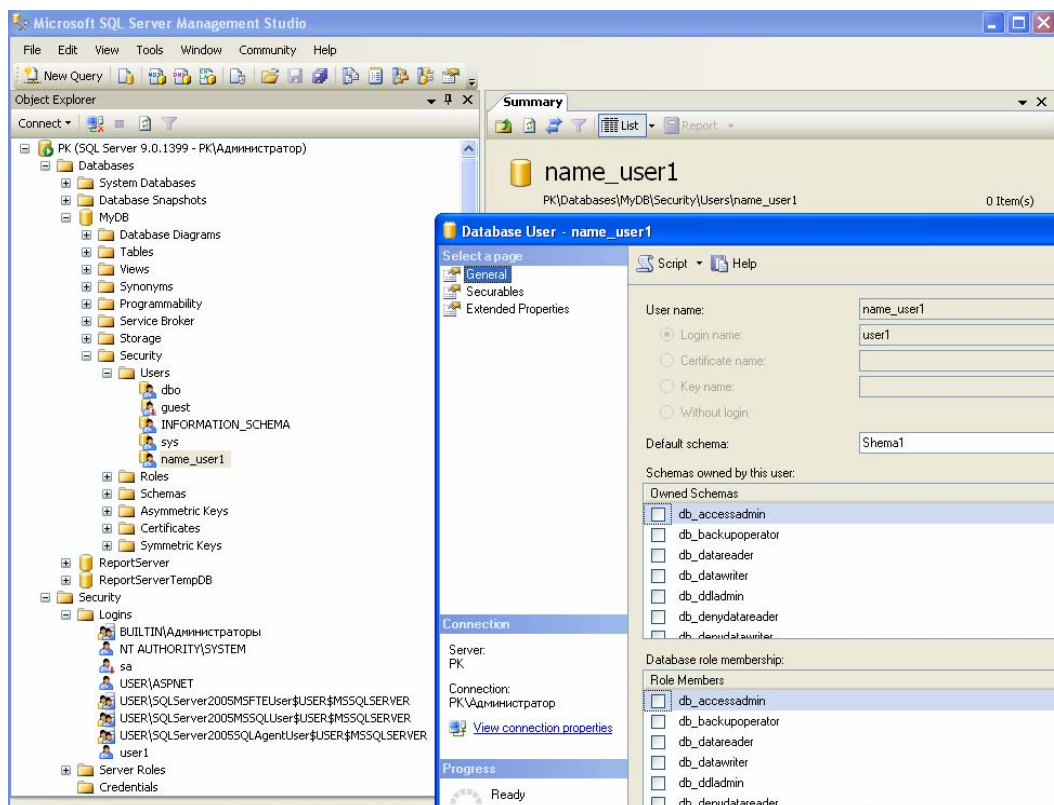
*Контекст выполнения* задается именем пользователя при входе (учетной записью пользователя), или при вызове выполняемого модуля.

Система безопасности строится на последовательном подключении к серверу SQL Server, а затем к самой базе данных. Базой данных по умолчанию первоначально устанавливается база данных master.

При аутентификации на уровне сервера проверяется учетная запись (login) и выполняется отображение данной учетной записи в имя пользователя базы данных (user). Все зарегистрированные учетные записи (login) отображаются в секции Security окна Object Explorer среды Microsoft SQL Server Management Studio, а пользователи каждой базы данных отображаются в секции Security соответствующей базы данных.

На следующем рисунке представлено соответствие между учетной записью user1 и именем пользователя базы данных MyDB name\_user1.





Можно говорить, что при подключение к базе данных происходит первичная (на уровне сервера) и вторичная (на уровне базы данных) идентификация пользователя.

SQL Server 2005 позволяет явно указать контекст выполнения таких определяемых пользователем модулей как: функции, процедуры, запросы и триггеры. Изменение контекста выполнения инициируется оператором EXECUTE, который имеет следующее формальное описание:

**Функции, хранимые процедуры и DML триггеры:**

```
{ EXEC | EXECUTE } AS { CALLER | SELF | OWNER |  
                        'ИМЯ_ПОЛЬЗОВАТЕЛЯ' }
```

**DDL триггеры уровня базы данных:**

```
{ EXEC | EXECUTE } AS { CALLER | SELF |  
                        'ИМЯ_ПОЛЬЗОВАТЕЛЯ' }
```

**DDL триггеры уровня сервера:**

```
{ EXEC | EXECUTE } AS { CALLER | SELF |  
                        'учетная_запись' }
```

**Запросы:**

```
{ EXEC | EXECUTE } AS { SELF | OWNER |  
                        'ИМЯ_ПОЛЬЗОВАТЕЛЯ' }
```

Ключевое слово CALLER определяет, что модуль выполняется в контексте пользователя, инициирующего выполнение. Для этого у данного пользователя должны быть соответствующие привелегии не только для запуска модуля, но и на доступ ко всем объектам базы данных, используемым в этом модуле. Значение CALLER используется как значение по умолчанию для всех модулей кроме запросов.

Указать конкретный контекст выполнения можно на стадии создания процедуры.

Например:

```
USE MyDB;  
GO
```

```
CREATE PROCEDURE dbo.test1 WITH EXECUTE AS 'user1'
AS
SELECT user_name(); -- Возвращает имя
                    -- текущего пользователя

GO
```

Далее применение установленного контекста выполнения указывается ключевым словом CALLER.

Например:

```
CREATE PROCEDURE dbo.test1 WITH EXECUTE AS 'name_user1'
AS
SELECT user_name(); -- Контекст выполнения установлен
                    -- на name_user1

EXECUTE AS CALLER;
SELECT user_name(); -- Контекст выполнения
                    -- установлен на name_user2,
                    -- выполнившего вызов модуля

REVERT;
SELECT user_name(); -- Контекст выполнения установлен
                    -- на name_user1

GO
```

Оператор REVERT выполняет переключение контекста выполнения на значение указанное в последнем выполненном операторе EXECUTE AS.

Для указания контекста выполнения сеанса используется оператор EXECUTE AS, который имеет следующее формальное описание:

```
{ EXEC | EXECUTE ] AS <спецификация_контекста>
[;]
```

где

```
< спецификация_контекста > ::=
```

```

{ LOGIN | USER } = 'name'
[ WITH { NO REVERT |
  COOKIE INTO @varbinary_variable } ]
| CALLER

```

## Создание таблицы

Для создания таблицы используется оператор **CREATE TABLE**, имеющий следующее формальное описание:

```

CREATE TABLE
[ имя_базы_данных . [схема] . |схема .] имя_таблицы
( { <определение_столбца> |
  <определение_вычислимого_столбца> }
[ <ограничение_целостности_таблицы> ] [ ,...n ] )
[ ON { partition_scheme_name (
  partition_column_name ) | filegroup
  | "default" } ]
[ { TEXTIMAGE_ON { filegroup | "default" } } ]
[ ; ]

<определение_столбца> ::=
имя_столбца <тип_данных>
[ COLLATE идентификатор_уппорядочивания ]
[ NULL | NOT NULL ]
[ [ CONSTRAINT идентификатор_ограничения_целостности ]
  DEFAULT выражение_ограничения_целостности ]
| [ IDENTITY [ ( нач_значение ,инкремент ) ]
  [ NOT FOR REPLICATION ]

```

```

]
[ ROWGUIDCOL ] [ <ограничение_целостности_столбца>
                                     [ ...n ] ]

<тип_данных> ::=
[ type_schema_name . ] type_name
  [ ( precision [ , scale ] | max |
    [ { CONTENT | DOCUMENT } ]
    xml_schema_collection ) ]

<ограничение_целостности_столбца> ::=
[ CONSTRAINT constraint_name ]
  { { PRIMARY KEY | UNIQUE }
    [ CLUSTERED | NONCLUSTERED ]
    [ WITH FILLFACTOR = fillfactor
    | WITH ( < index_option > [ , ...n ] )
  }
[ ON { partition_scheme_name ( partition_column_name )
    | filegroup | "default" } ]
| [ FOREIGN KEY ] REFERENCES [ схема . ]
    referenced_table_name [ ( ref_column ) ]
[ ON DELETE { NO ACTION | CASCADE
  | SET NULL | SET DEFAULT } ]
[ ON UPDATE { NO ACTION | CASCADE | SET NULL
  | SET DEFAULT } ]
[ NOT FOR REPLICATION ]
| CHECK [ NOT FOR REPLICATION ] ( лог_выражение )
}

```

```

<определение_вычислимого_столбца> ::=
    имя_столбца AS вычисляемое_выражение
    [ PERSISTED [ NOT NULL ] ]
    [ [ CONSTRAINT constraint_name ]
      { PRIMARY KEY | UNIQUE }
      [ CLUSTERED | NONCLUSTERED ]
      [ WITH FILLFACTOR = fillfactor
        | WITH ( <index_option> [ , ...n ] )
      ]
      | [ FOREIGN KEY ] REFERENCES referenced_table_name
        [ ( ref_column ) ]
        [ ON DELETE { NO ACTION | CASCADE } ]
        [ ON UPDATE { NO ACTION } ]
        [ NOT FOR REPLICATION ]
      | CHECK [ NOT FOR REPLICATION ] ( logical_expression )
      [ ON { partition_scheme_name ( partition_column_name )
        | filegroup | "default" } ]
    ]

< ограничение_целостности_таблицы > ::=
    [ CONSTRAINT constraint_name ]
    { { PRIMARY KEY | UNIQUE }
      [ CLUSTERED | NONCLUSTERED ]
        (column [ ASC | DESC ] [ ,...n ] )
      [ WITH FILLFACTOR = fillfactor
        | WITH ( <index_option> [ , ...n ] )
      ]
      [ ON { partition_scheme_name
        (partition_column_name)

```

```

        | filegroup | "default" } ]
    | FOREIGN KEY ( column [ ,...n ] )
        REFERENCES referenced_table_name [ (
            ref_column [ ,...n ] ) ]
    [ ON DELETE { NO ACTION | CASCADE | SET NULL |
        SET DEFAULT } ]
    [ ON UPDATE { NO ACTION | CASCADE | SET NULL |
        SET DEFAULT } ]
    [ NOT FOR REPLICATION ]
    | CHECK [ NOT FOR REPLICATION ] (лог_выражение )
}

<опции_индекса> ::=
{
    PAD_INDEX = { ON | OFF }
    | FILLFACTOR = fillfactor
    | IGNORE_DUP_KEY = { ON | OFF }
    | STATISTICS_NORECOMPUTE = { ON | OFF }
    | ALLOW_ROW_LOCKS = { ON | OFF}
    | ALLOW_PAGE_LOCKS = { ON | OFF}
}

```

При создании таблицы может быть указана база данных и схема, в которую будут добавлена таблица. Описание столбцов указывается в круглых скобках через запятую. Столбцы могут быть двух видов: столбцы заданного типа и вычисляемые столбцы.

При создании таблицы можно определить ее разбиение.

Ограничения целостности для столбца могут указываться следующими фразами:

- NOT NULL – в любой добавляемой или изменяемой строке столбец всегда должен иметь значение отличное от NULL;
- UNIQUE – все значения столбца должны быть уникальны;
- PRIMARY KEY – устанавливает один столбец как первичный ключ и одновременно подразумевает что все значения столбца будут уникальны;
- CLUSTERED – определяет создание кластерного индекса;
- NONCLUSTERED – определяет создание некластерного индекса;
- CHECK (condition) – указываемое в скобках условие используется для сравнения значение столбца и возвращает TRUE, FALSE или UNKNOWN. Если при попытке выполнения SQL-оператора возвращаемое значение равно FALSE, то оператор выполнен не будет;
- REFERENCES table (fields\_list) – ограничение требует совпадения значений столбцов данной таблицы с указанными столбцами родительской таблицы.

Ограничения целостности для таблицы могут указываться следующими фразами:

- UNIQUE (column) – все значения столбца должны быть уникальны;
- PRIMARY KEY(column) – устанавливает один столбец как первичный ключ и одновременно подразумевает что все значения столбца будут уникальны;
- CLUSTERED (column) – определяет создание кластерного индекса;
- NONCLUSTERED (column) – определяет создание некластерного индекса;
- CHECK (condition) – указываемое в скобках условие используется для сравнения значение столбца и возвращает TRUE, FALSE или UNKNOWN. Если при попытке выполнения SQL-оператора возвращаемое значение равно FALSE, то оператор выполнен не будет;



- FOREIGN KEY (fields\_list) – это ограничение по внешнему ключу для столбцов гарантирует, что все значения, указанные во внешнем ключе будут соответствовать значениям родительского ключа, обеспечивая ссылочную целостность;
- REFERENCES table (fields\_list) – ограничение требует совпадения значений столбцов данной таблицы с указанными столбцами родительской таблицы.

Таблицы могут быть постоянными и временными. В свою очередь временные таблицы делятся на локальные - доступные только в рамках данного сеанса, и глобальные – доступные во всех сеансах. Имя локальной таблицы начинается с префикса #, а имя глобальной – с префикса ##.

### Формирование запросов

Извлечение информации из базы данных выполняется посредством оператора запроса SELECT, который имеет следующее формальное описание:

#### SELECT

```
[WITH <common_table_expression> [,...n]]
  <выражение запроса>
[ ORDER BY { выражение_для_упорядочивания
              | номер_столбца [ ASC | DESC ] }
              [ ,...n ] ]
[ COMPUTE
  { { AVG | COUNT | MAX | MIN | SUM } ( выражение ) }
              [ ,...n ]
              [ BY выражение [ ,...n ] ]
]
[ FOR { BROWSE | <XML> } ]
[ OPTION ( <query_hint> [ ,...n ] ) ]
```

**<выражение\_запроса> ::=**

```
{ <спецификация_запроса> | ( <выражение_запроса> ) }
[ { UNION [ ALL ] | EXCEPT | INTERSECT }
  <спецификация_запроса> | ( <выражение_запроса> )
  [...n ] ]
```

**<спецификация\_запроса> ::=**

```
SELECT [ ALL | DISTINCT ]
  [ TOP выражение [ PERCENT ] [ WITH TIES ] ]
  < список_выбора >
  [ INTO новая_таблица ]
  [ FROM { <исходная_таблица> } [ ,...n ] ]
  [ WHERE <условие> ]
  [ GROUP BY [ ALL ] выражение_используемое_для_группы
    [ ,...n ] ]

  [ WITH { CUBE | ROLLUP } ]
  ]
  [ HAVING < условие > ]
```

**<XML>::=**

```
XML
{{ RAW [ ( 'ElementName' ) ] | AUTO }
  [
    [, BINARY BASE64 ][, TYPE ][, ROOT [ ('RootName')]]
    [, { XMLDATA | XMLSCHEMA [ ('TargetNameSpaceURI')]] ]
    [, ELEMENTS [ XSINIL | ABSENT ]
  ]
  | EXPLICIT
  [
    [, BINARY BASE64 ][, TYPE ][, ROOT [ ('RootName')]]
    [ , XMLDATA ]
```

```

]
| PATH [ ( 'ElementName' ) ]
[
  [, BINARY BASE64 ][, TYPE ][, ROOT [ ('RootName')]]
  [ , ELEMENTS [ XSINIL | ABSENT ] ]
]
}

```

Если оператор SELECT выполняется из приложения на другом языке программирования, то формируется результирующий набор, размещаемый в памяти приложения или сервера БД, а затем приложение извлекает данные из результирующего набора в свои переменные.

После фразы SELECT указывается список выражений, определяющий значения формируемые запросом. В самом простом случае список выражений является списком полей таблицы. Если требуется извлечение значений всех полей, то вместо списка полей можно указать символ \*.

Например: SELECT \* FROM tbl1; .

Имя поля может быть квалифицировано именем таблицы, указываемым через точку. Например: SELECT tbl1.f1, tbl2.f1 FROM tbl1, tbl2; .

Фраза **FROM** определяет одну или несколько таблиц или подзапросов, используемых для извлечения данных.

Фраза **WHERE** определяет условие, которому должны удовлетворять все строки, используемые для формирования результирующего набора. Предикат содержит одно или несколько выражений, выполняющих сравнения. В выражениях могут участвовать имена столбцов, функции агрегирования, переменные.

Кроме стандартных операторов сравнения, таких как =, <>, >, <, >=, <= в предикате могут быть использованы операторы такие, как:

- **BETWEEN** – возвращает TRUE, если значение находится в указанном диапазоне. Например:  $x \text{ BETWEEN } y \text{ AND } z$  эквивалентно выражению  $(x \leq z) \text{ AND } (x \geq y)$ ;
- **IN** - совпадает с одним из перечисленных в списке. Например:  $x \text{ IN } (a, b, c)$ ;
- **LIKE** - возвращает TRUE для значений, совпадающих с указанной подстрокой символов. Например:  $x \text{ LIKE 'abc'}$ ;
- **IS NULL** - возвращает TRUE, если значение равно NULL. Этот предикат возвращает только значение TRUE или FALSE Например:  $x \text{ IS NULL}$ ;
- **EXISTS** – это предикат существования, возвращающий значение TRUE, если указанный в нем подзапрос содержит хотя бы одну строку. Например:  

```
SELECT * FROM tbl1 t_out
      WHERE EXISTS (SELECT * FROM tbl1 t_in
                    WHERE t_in.f1 = t_out.f1);
```

Фраза **GROUP BY** оператора SELECT применяется для определения группы строк, над которыми выполняются функции агрегирования. Если в операторе SELECT указана фраза GROUP BY, то все имена столбцов, указываемые в списке для определения создаваемого результирующего набора, должны быть указаны с функциями агрегирования. Так как для каждой группы строк в результирующий набор будет включена только одна строка, содержащая значения полученные функциями агрегирования над данной группой строк.

К функциям агрегирования относятся следующие функции языка Transact-SQL:

- **COUNT** ( { [ [ ALL | DISTINCT ] выражение ] | \* } ) – подсчет количества значений столбцов (ALL- всех для заданного выражения, DIST – учитывая только уникальные не равные NULL, \* - подсчет всех строк);

- **AVG** – определение среднего значения;
- **SUM** – подсчет суммы всех значений группы. Если при этом получаемое значение выходит за пределы суммируемого типа данных, то инициируется ошибка выполнения SQL-оператора;
- **MAX** - определение максимального значения из группы;
- **MIN** - определение минимального значения из группы.

Фраза **HAVING** оператора **SELECT** определяет предикат аналогично фразе **WHERE**, но применяемый к строкам, полученным в результате выполнения функций агрегирования.

Следующий пример иллюстрирует применение групп. Столбец **id1** имеет всего три различных значения: 11, 22 и 33. Для каждой из трех групп находится минимальное и максимальное значение столбца **f1**.

```
SELECT id1, MIN(f1), MAX(f1)
FROM tbl1
GROUP BY id1;
```

Результатом выполнения этого SQL-оператора будет формирование следующих строк:

id1	MIN (f1)	MAX (f1)
11	125	600
22	200	2300
33	100	450

При выборе с применением групп и с дополнительным ограничением на значение в столбце **MIN(f1)**.

```
SELECT dno, MIN(f1), MAX(f1)
FROM tbl1
GROUP BY id1
HAVING MAX(f1) < 1000;
```

В результате выполнения этого SQL-оператора будут возвращены только две строки: первая и последняя:

id1	MIN (f1)	MAX (f1)
-----	----------	----------

```
-----
11      125      600
33      100      450
```

Фраза FOR XML оператора SELECT определяет, что результат запроса будет возвращен в виде XML-документа.

## Соединение таблиц

Для соединения таблиц с одноименными столбцами или соединении таблицы с самой собой используются алиасы, задаваемые во фразе FROM через фразу AS после имени таблицы.

Например:

```
select t1.f1, t1.f2, t2.f1, t2.f2 from tbl1 as t1, tbl1 as t2 where t1.f1= t2.f2;
```

Соединение таблиц определяется фразой FROM оператора SELECT, имеющей следующее формальное описание:

```
[ FROM { <исходная_таблица> } [ , ...n ] ]
<исходная_таблица> ::=
{ имя_таблицы_или_представления [ [ AS ] алиас_таблицы ]
  [ TABLESAMPLE [SYSTEM] ( sample_number [ PERCENT |
                                                                    ROWS ] )
    [REPEATABLE ( repeat_seed )]
  ]
  [ WITH ( < table_hint > [ [ , ]...n ] ) ]
  | rowset_function [ [ AS ] алиас_таблицы ]
  [ ( bulk_column_alias [ ,...n ] ) ]
  | user_defined_function [ [ AS ] алиас_таблицы]
  | OPENXML <openxml_clause>
  | derived_table [ AS ] алиас_таблицы
```

```

[ ( алиас_столбца [ , ...n ] ) ]
| <соединяемые_таблицы>
| <pivoted_table>
| <unpivoted_table>
}

<соединяемые_таблицы> ::=
{
  <исходная_таблица> <тип_соединения> <исходная_таблица>
                                ON <условие>
| <исходная_таблица> CROSS JOIN <исходная_таблица>
| левая_исходная_таблица { CROSS | OUTER } APPLY
                                правая_исходная_таблица
| [ ( ] <соединяемые_таблицы> [ ) ]
}

<тип_соединения> ::=
[ { INNER | { { LEFT | RIGHT | FULL } [ OUTER ] } }
  [ <join hint> ] ]
JOIN

<pivoted_table> ::=
  исходная_таблица PIVOT <pivot_clause> table_alias

<pivot_clause> ::=
  ( функция_агрегирования ( value_column )
    FOR pivot_column IN ( <column_list> )
  )

<unpivoted_table> ::=

```

```
исходная_таблица UNPIVOT <unpivot_clause> алиас_таблицы

<unpivot_clause> ::=
    (value_column FOR pivot_column IN (<список_столбцов>))

<список_столбцов> ::=
    имя_столбца [ , ... ]
```

Если фраза FROM определяет более одной таблицы или подзапроса, то все эти таблицы соединяются. Перекрестное соединение (CROSS JOIN), называемое также декартовым произведением создает результирующий набор со всеми возможными комбинациями строк.

Соединения позволяют выполнять временное объединение данных, не предусмотренное схемой (родительскими и внешними ключами).

Соединяемые таблицы перечисляются через запятую во фразе FROM оператора SELECT.

Во фразе FROM можно использовать следующие типы соединений:

- CROSS JOIN - перекрестное соединение;
- INNER JOIN – внутреннее соединение. Это соединение используется по умолчанию. При внутреннем соединении в результирующий набор включаются только те строки, значения которых по соединяемым (одноименным) столбцам совпадают;
- LEFT [OUTER] JOIN – левое внешнее соединение. При внешнем левом соединении в результирующий набор будут выбраны все строки из левой таблицы (указываемой первой). При совпадении значений по соединяемым (одноименным) столбцам значения второй таблицы заносятся в результирующий набор в соответствующие строки. При отсутствии совпадений в качестве значений второй таблицы проставляется значение NULL;



- **RIGHT [OUTER] JOIN** – правое внешнее соединение. При внешнем правом соединении в результирующий набор будут выбраны все строки из правой таблицы (указываемой второй). При совпадении значений по соединяемым (одноименным) столбцам значения первой таблицы заносятся в результирующий набор в соответствующие строки (рис. 3.6.). При отсутствии совпадений в качестве значений первой таблицы проставляется значение NULL;
- **FULL [OUTER] JOIN** - полное внешнее соединение. При полном внешнем соединении в результирующий набор будут выбраны все строки, как из правой, так и из левой таблицы. При совпадении значений по соединяемым (одноименным) столбцам строка содержит значения и из левой и из правой таблицы. В противном случае, вместо отсутствующих значений в столбцы таблицы (левой или правой) заносится значение NULL.

Фраза **ON** позволяет выполнить естественное соединение по указываемому предикату. В результирующий набор выбираются строки, удовлетворяющие заданному условию. Этот способ соединения аналогичен соединению по предикату, указываемому фразой **WHERE**.

### **Подзапросы**

Спецификация запроса, указываемая в операторах языка Transact-SQL, описывает *подзапрос*. Подзапрос является очень мощным средством языка SQL, позволяя строить сложные иерархии запросов, многократно выполняемые в процессе построения результирующего набора или выполнения одного из операторов изменения данных (**DELETE**, **INSERT**, **UPDATE**).

Условно подзапросы иногда подразделяют на три типа, каждый из которых является сужением предыдущего:

- табличный подзапрос, возвращающий набор строк и столбцов;
- подзапрос строки, возвращающий только одну строку, но возможно несколько столбцов;

- скалярный подзапрос, возвращающий значение одного столбца в одной строке.

Подзапрос позволяет решать следующие задачи:

- определять набор строк, добавляемый в таблицу на одно выполнение оператора INSERT;
- определять данные, включаемые в представление, создаваемое оператором CREATE VIEW.
- для определения значений, модифицируемых оператором UPDATE;
- для указания одного или нескольких значений во фразах WHERE и HAVING оператора SELECT;
- для определения во фразе FROM таблицы как результата выполнения подзапроса;

Например:

```
SELECT * INTO t1
```

```
FROM (select * from tbl1 WHERE f2 LIKE 'aa%')
```

- для применения коррелированных подзапросов. Подзапрос называется коррелированным, если запрос, содержащийся в предикате, имеет ссылку на значение из таблицы (внешней к данному запросу), которая проверяется посредством данного предиката.

## Объединения

Язык Transact-SQL предоставляет два способа объединения таблиц:

- указывая соединяемые таблицы (включая подзапросы) во фразе **FROM** оператора SELECT. В этом случае сначала выполняется соединение таблиц, а уже потом к полученному множеству применяются условия, указанные фразой WHERE, агрегирование, определяемое фразой GROUP BY, упорядочивание данных и т.п.;

- определяя объединение результирующих наборов, полученных при обработке оператора SELECT. В этом случае два оператора SELECT соединяются фразой **UNION**, **INTERSECT** или **EXCEPT**.

Для объединения результатов нескольких запросов в один результирующий набор в операторе SELECT используется фраза UNION:

```
{ <спецификация_запроса> | ( <выражение_запроса> ) }  
  UNION [ ALL ]  
<спецификация_запроса> | ( <выражение_запроса> )  
  [ UNION [ ALL ]  
<спецификация_запроса> | ( <выражение_запроса> )  
  [ ...n ] ]
```

Фраза UNION объединяет результаты двух запросов по следующим правилам:

- каждый из объединяемых запросов должен содержать одинаковое число столбцов;
- тип значений из попарно объединяемых столбцов должен быть одинаковым или приводимым. Так нельзя объединять значения из столбца типа integer и столбца типа varchar;
- из результирующего набора автоматически исключаются совпадающие строки.

Фраза UNION ALL выполняет объединение двух подзапросов аналогично фразе ALL со следующими исключениями:

- совпадающие строки не удаляются из формируемого результирующего набора;
- объединяемые запросы выводятся в результирующем наборе последовательно без упорядочивания.

При объединении более двух запросов для изменения порядка выполнения операции объединения можно использовать скобки

Фраза **INTERSECT** позволяет выбрать только те строки, которые присутствуют в каждом объединяемом результирующем наборе.

Фраза **EXCEPT** позволяет выбрать только те строки, которые присутствуют в первом объединяемом результирующем наборе, но отсутствуют во втором результирующем наборе.

При выполнении объединения запросов, указываемых фразами **UNION**, **EXCEPT** и **INTERSECT**, существуют следующие ограничения:

- фразу **INTO** может содержать только первый подзапрос;
- фраза **ORDER BY** применяется только ко всему оператору (указывается в конце), применение этой фразы к любому подзапросу, входящему в объединение не допускается;
- фразы **GROUP BY** и **HAVING** могут применяться только к подзапросам, применять их к получаемому конечному результирующему набору нельзя;
- фраза **FOR BROWSE** не может быть указана в операторах, использующих фразы **UNION**, **EXCEPT** и **INTERSECT**.

В операторе **INSERT** можно использовать фразы **UNION**, **EXCEPT** и **INTERSECT**.

### **Операторы управления данными**

К операторам управления данными относятся операторы **UPDATE**, **INSERT** и **DELETE**.

Оператор **UPDATE** выполняет изменение данных таблицы или представления и имеет следующее формальное описание:

```
[ WITH <common_table_expression> [...n] ]  
UPDATE  
  [ TOP ( выражение ) [ PERCENT ] ]  
  { <объект> | rowset_function_limited  
    [ WITH ( <Table_hint_Limited> [ ...n ] ) ]
```

```

    }
    SET { имя_столбца = { выражение | DEFAULT | NULL }
        | {udt_column_name.{ { property_name = expression
                               | field_name = expression }
                               | method_name (argument [ ,...n ] )
                              }
        }
    | имя_столбца { .WRITE (выражение ,@Offset ,@Length) }
    | @variable = expression
    | @variable = column = expression [ ,...n ]
    } [ ,...n ]
    [ <фраза_OUTPUT> ]
    [ FROM{ <table_source> } [ ,...n ] ]
    [ WHERE { <search_condition>
              | { [ CURRENT OF
                  { { [ GLOBAL ] cursor_name }
                    | cursor_variable_name
                  }
                ]
              }
    ]
    [ OPTION ( <query_hint> [ ,...n ] ) ]
[ ; ]

<объект> ::=
{
    [ сервер . база_данных . схема .
    | база_данных .[ схема ] .
    | схема .

```

] <i>имя_таблицы_или_представления</i>
--

Оператор INSERT применяется для добавления строк в таблицу или представление и имеет следующее формальное описание:

<pre> [ WITH &lt;common_table_expression&gt; [ ,...n ] ] <b>INSERT</b>   [ TOP ( <i>выражение</i> ) [ PERCENT ] ]   [ INTO]   { &lt;объект&gt;   <i>rowset_function_limited</i>     [ WITH ( &lt;Table_Hint_Limited&gt; [ ...n ] ) ]   } {   [ ( <i>column_list</i> ) ]   [ &lt;фраза_OUTPUT&gt; ]   { <b>VALUES</b> ( { DEFAULT   NULL   <i>выражение</i> } [ ,...n ] )       <i>derived_table</i>       <i>execute_statement</i>   } }     DEFAULT VALUES [; ] </pre>
---

Оператор DELETE выполняет удаление строк из таблицы или представления и имеет следующее формальное описание:

<pre> [ WITH &lt;common_table_expression&gt; [ ,...n ] ] <b>DELETE</b>   [ TOP ( <i>выражение</i> ) [ PERCENT ] ]   [ <b>FROM</b> ] </pre>
--

```

{ <объект> | rowset_function_limited
  [ WITH ( <table_hint_limited> [ ...n ] ) ]
}
[ <фраза_OUTPUT> ]
[ FROM <table_source> [ ,...n ] ]
[ WHERE { <search_condition>
  | { [ CURRENT OF
      { { [ GLOBAL ] имя_курсора }
        | имя_переменной_курсора
      }
    ]
  }
}
]
[ OPTION ( <Query Hint> [ ,...n ] ) ]
[; ]

```

Фраза WITH <common\_table\_expression> определяет временный именуемый результирующий набор, используемый другими операторами модуля.

Например:

```

USE MYDB;
WITH NewRes(f1, f3) AS
( SELECT f1, COUNT(*)
  FROM Tbl1 AS t1
  WHERE f1 IS NOT NULL
  GROUP BY f1
)
SELECT f1, f3 FROM NewRes ORDER BY f1;

```

В поле f3 запроса из NewRes отображается количество строк в группе, создаваемой по полю f1.



## Тема 3. Программная среда SQL-сервера.

### Язык Transact-SQL

Язык Transact-SQL позволяет записывать выражения, объявлять константы и переменные, реализовывать процедуры.

Объявление переменных выполняется оператором DECLARE. Имя объявляемой переменной начинается с символа @.

Язык Transact-SQL позволяет использовать большой набор типов данных (включая типы данных, присущие Microsoft SQL Server, специальные типы данных (uniqueidentifier, sql\_variant) и тип XML).

### Типы данных

Типы данных имеют не только переменные, но и такие объекты как: столбцы таблиц и представлений; параметры хранимых процедур; возвращаемые значения для функций.

В языке Transact-SQL поддерживаются следующие типы данных:

#### ■ целочисленные:

int	от $-2^{31}$ (-2,147,483,648) до $2^{31}-1$ (2,147,483,647)	4 байта
smallint	$-2^{15}$ (-32,768) to $2^{15}-1$ (32,767)	2 байта
tinyint	от 0 до 255	1 байт
bigint	от $-2^{63}$ (-9,223,372,036,854,775,808) to $2^{63}-1$ до (9,223,372,036,854,775,807)	8 байт
bit	1, 0 или NULL	

#### ■ вещественные с фиксированной точностью:

numeric	максимально допустимое число знаков 38
decimal	максимально допустимое число знаков 38

#### ■ вещественные с плавающей точкой:

float	- 1.79E+308 to -2.23E-308, 0 and 2.23E-308 to 1.79E+308
-------	---

float [ ( n ) ] – n число битов, используемое мантиссой (от 1 до 53); в зависимости от этого значение занимает 4 байта (от 1 до 24, точность 7 знаков) или 8 байтов (25-53, точность 15 знаков)

real - 3.40E + 38 to -1.18E - 38, 0 and 1.18E - 38 to 3.40E + 38

▪ **денежный**

money от -922,337,203,685,477.5808 до 922,337,203,685,477.5807 8 байтов

smallmoney от - 214,748.3648 до 214,748.3647 4 байта

▪ **символьные:**

char

varchar

text\*

▪ **символьные, использующие Unicode:**

nchar

ntext\*

nvarchar

▪ **двоичные:**

binary

varbinary

image\*

▪ **типы даты/времени:**

datetime

smalldatetime

▪ **для работы с курсором**

cursor (можно использовать только для переменных и параметров хранимых процедур)

▪ **для хранения результирующего набора:**

table

▪ **остальные типы:**

sql\_variant – для хранения переменных любого типа (за некоторым исключением)

uniqueidentifier – для хранения 16 байтового GUID

xml – для данных в XML-формате

timestamp – синоним тип rowversion.

Типы помеченные символом \* не предполагается поддерживать в последующих версиях Microsoft SQL Server.

Transact-SQL позволяет применять два типа идентификаторов: стандартные и ограниченные (требуют заключения в квадратные скобки).

Для доступа к объектам базы данных из модулей Transact-SQL применяются правила классификации

имен(имя\_сервера.имя\_базы\_данных.владелец.имя\_объекта).

## **Операторы языка Transact-SQL**

Язык Transact-SQL поддерживает следующие конструкции:

- блок:  
BEGIN           END,
- условный оператор:  
IF Boolean\_expression  
    { sql\_statement | statement\_block }  
[ ELSE  
    { sql\_statement | statement\_block } ]
- цикл WHILE:  
WHILE Boolean\_expression  
    { sql\_statement | statement\_block }  
    [ BREAK ]  
    { sql\_statement | statement\_block }  
[ CONTINUE ]  
    { sql\_statement | statement\_block }
- функция CASE:  
CASE input\_expression  
    WHEN when\_expression THEN result\_expression

- ```
        [ ...n ]
        [
        ELSE else_result_expression
        ]
    END
```
- функция CASE:  
CASE  
 WHEN Boolean\_expression THEN result\_expression  
 [ ...n ]  
 [
 ELSE else\_result\_expression
 ]  
 END
  - функция COALESCE.

Язык Transact-SQL предоставляет набор встроенных функций (математические, строковые, функции даты/времени, функции конфигурирования, системные функции, функции системы безопасности, функции управления метаданными, статические данные и т.п.) и предоставляет возможность создавать *функции, определяемые пользователем*.

В языке Transact-SQL можно использовать кросстабулированные данные посредством фраз PIVOT и UNPIVOT, значительно упрощающих код операторов запроса.

Для добавления данных предоставляется оператор MERGE.

Язык Transact-SQL позволяет выполнять обработку исключений посредством оператора TRY-CATCH.

## Модули

Для создания хранимых процедур используется оператор CREATE PROCEDURE, который имеет следующее формальное описание:

```

CREATE { PROC | PROCEDURE } [схема.] имя_процедуры [ ;
number ]
    [ { @параметр [ схема_тип_данных. ] тип_данных }
      [ VARYING ] [ = default ] [ [ OUT [ PUT ]
    ] [ ,...n ]
  [ WITH
      [ ENCRYPTION ]
      [ RECOMPILE ]
      [ фраза_EXECUTE_AS ]
  [ FOR REPLICATION ]
  AS { {[ BEGIN ] операторы [ END ]} [;][ ...n ] |
  EXTERNAL NAME сборка.класс.метод
  }
  [;]

```

Для создания функций используется оператор **CREATE FUNCTION**, который имеет следующее формальное описание:

**Скалярная функция:**

```

CREATE FUNCTION [ схема. ] имя_функции
( [ { @параметр [ AS ][ схема_типа_данных.
] тип_парамета
    [ = default ] }
  [ ,...n ]
]
)
  RETURNS тип_возвращаемого_значения
  [ WITH <function_option> [ ,...n ] ]

```

```
[ AS ]
BEGIN
    тело функции
    RETURN возвращаемое значение
END
[ ; ]

Inline функция:
CREATE FUNCTION [ схема. ] имя_функции
( [ { @параметр [ AS ] [ схема_типа_данных.
] тип_параметра
    [ = default ] }
    [ ,...n ]
]
)
RETURNS TABLE
    [ WITH <опции_функции> [ ,...n ] ]
    [ AS ]
    RETURN [ ( ] оператор_select [ ) ]
[ ; ]

Функция, возвращающая таблицу:
CREATE FUNCTION [ схема. ] имя_функции
( [ { @параметр [ AS ] [ схема_типа_данных.
] тип_параметра
    [ = default ] }
    [ ,...n ]
]
)
)
```

```

RETURNS @return_variable TABLE <определение_таблицы>
    [ WITH <опции_функции> [ ,...n ] ]
    [ AS ]
BEGIN
    тело_функции
RETURN
END
[ ; ]

CLR функция:
CREATE FUNCTION [ schema_name. ] function_name
( { @parameter_name [AS] [ type_schema_name. ]
parameter_data_type
    [ = default ] }
    [ ,...n ]
)
RETURNS { return_data_type | TABLE
<clr_table_type_definition> }
    [ WITH <clr_function_option> [ ,...n ] ]
    [ AS ]
    EXTERNAL NAME assembly_name.class_name.method_name
[ ; ]

<опции_функции> ::=
{ [ ENCRYPTION ]
| [ SCHEMABINDING ]
| [ RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT ]
| [ EXECUTE_AS_фраза ]
}

```

```

<опции_clr_функции>::=
{
    [ RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT ]
    | [ EXECUTE_AS_фраза ]
}

<определение_таблицы>::=
( { <определение_столбца>
  <ограничения_целостности_столбца>
    | <определение_вычислимого_столбца> }
  [ <ограничения_целостности_таблицы> ] [ ,...n ]
)

<clr_определение_таблицы>::=
( { столбец тип_данных } [ ,...n ] )

<определение_столбца>::=
{
    { столбец тип_данных }
    [ [ DEFAULT constant_expression ]
      [ COLLATE collation_name ] | [ ROWGUIDCOL ]
    ]
    | [ IDENTITY [ (нач_значение , приращение ) ] ]
    [ <ограничение_целостности_столбца> [ ...n ] ]
}

<ограничение_целостности_столбца>::=
{ [ NULL | NOT NULL ]
  { PRIMARY KEY | UNIQUE }
  [ CLUSTERED | NONCLUSTERED ]
}

```



```

        [ WITH FILLFACTOR = fillfactor
        | WITH ( < опции_индекса > [ , ...n ] )
        [ ON { filegroup | "default" } ]
        | [ CHECK ( logical_expression ) ] [ ,...n ]
    }

<определение_вычислимого_столбца>::=
столбец AS выражение

<ограничение_целостности_таблицы>::=
{
    { PRIMARY KEY | UNIQUE }
    [ CLUSTERED | NONCLUSTERED ]
    ( столбец [ ASC | DESC ] [ ,...n ] )
    [ WITH FILLFACTOR = fillfactor
    | WITH ( <опции_индекса> [ , ...n ] )
    | [ CHECK ( logical_expression ) ] [ ,...n ]
}

<опции_индекса>::=
{ PAD_INDEX = { ON | OFF }
  | FILLFACTOR = fillfactor
  | IGNORE_DUP_KEY = { ON | OFF }
  | STATISTICS_NORECOMPUTE = { ON | OFF }
  | ALLOW_ROW_LOCKS = { ON | OFF }
  | ALLOW_PAGE_LOCKS = { ON | OFF }
}

```

Максимально допустимое количество параметров для функции – 1024. Имя параметра функции начинается с префикса @.

Следующий пример иллюстрирует применение inline-функции, возвращающей таблицу:

```
CREATE FUNCTION S1.MyFn (@i1 int)
RETURNS TABLE
AS
RETURN
( SELECT t1.ID, t1.Name, SUM(t2.F3) AS 'Поле суммы'
  FROM tbl1 AS t1
    JOIN tbl2 AS t2 t2.ID = t1.ID
    JOIN tbl3 AS t3 ON t3.IDzak = t2.IDzak
 WHERE t3.IDcust = @i1
 GROUP BY t1.ID, t1.Name
);
GO
```

Значения функций, возвращающих таблицу, могут быть использованы в SQL-операторах таким же образом, как и подзапросы.

Например, вызов функции может быть указан во фразе FROM оператора SELECT следующим образом:

```
SELECT * FROM S1.MyFn (55); -- параметр указывает
                             -- значение поля IDcust в tbl3
```

## Тема 4. Механизмы удаленного доступа. Интерфейс ODBC.

### Архитектура ODBC

Интерфейс ODBC (Open Database Connectivity) был разработан фирмой Microsoft как открытый интерфейс доступа к базам данных, предоставляющий унифицированные средства взаимодействия прикладной

программы, называемой клиентом (или приложением-клиентом) с сервером – базой данных.

В основу интерфейса ODBC были положены спецификация CLI-интерфейса (Call-Level Interface), разработанная X/Open, и ISO/IEC для API баз данных, и язык SQL (Structured Query Language), как стандарт языка доступа к базам данных.

Интерфейс ODBC проектировался для поддержки максимальной интероперабельности приложений, обеспечивающей унифицированный доступ любого приложения, использующего ODBC, к различным источникам данных.

Для взаимодействия с базой данных приложение-клиент вызывает функции интерфейса ODBC, которые реализованы в специальных модулях, называемых ODBC-драйверами. Как правило, ODBC-драйверы это DLL-библиотеки. При этом одна DLL-библиотека может поддерживать несколько ODBC-драйверов. При установке на компьютер любого SQL-сервера автоматически выполняется регистрация в реестре Windows и соответствующего ODBC-драйвера.

Архитектура ODBC представляется четырьмя компонентами:

- *Приложение-клиент*, выполняющее вызов функций ODBC.
- *Менеджер драйверов*, загружающий и освобождающий ODBC-драйверы, требуемые приложениям-клиентам. Менеджер драйверов обрабатывает вызовы ODBC-функций или передает их драйверу.
- *ODBC-драйвер*, обрабатывающий вызовы SQL-функций, передавая SQL-серверу выполняемый SQL-оператор, а приложению-клиенту – результат выполнения вызванной функции.
- *Источник данных*, определяемый как конкретная локальная или удаленная база данных.

Основное назначение менеджера драйверов – загрузка драйвера, соответствующего подключаемому источнику данных и инкапсуляция взаимодействия с различными типами источников данных посредством применения различных ODBC-драйверов.

### **Источник данных**

Для подключения к SQL-серверу с использованием функций ODBC API первоначально следует создать источник данных DSN, в котором определяется используемое имя источника данных и местоположение базы данных (в зависимости от конкретной БД это может быть имя файла базы данных или имя сервера и имя базы данных).

Создать источник данных можно как программно - вызвав функцию ODBC API, так и интерактивно – используя утилиту ODBC.

DLL-библиотека ODBC32.DLL предоставляет функции ODBC API ConfigDSN и SQLConfigDataSource, позволяющие выполнять регистрацию новых источников данных или удалять информацию об источниках данных из реестра Windows.

Для использования в среде Visual Studio функций ConfigDSN и SQLConfigDataSource следует подключить заголовочный файл odbinst.h.

Функция ConfigDSN позволяет добавлять, изменять или удалять источники данных и имеет следующее формальное описание:

```
BOOL ConfigDSN(  
    HWND      hwndParent,  
    WORD      fRequest,  
    LPCSTR     lpszDriver,  
    LPCSTR     lpszAttributes);
```

. Параметр *fRequest* указывает тип запроса, который задается одной из следующих констант:

- ODBC\_ADD\_DSN – добавление нового источника данных;
- ODBC\_CONFIG\_DSN – изменение существующего источника данных;

- ODBC\_REMOVE\_DSN – удаление существующего источника данных.

Параметр *lpszDriver* содержит описание драйвера, а параметр *lpszAttributes* – список атрибутов в форме "ключевое слово=значение" (например: DSN=MyDB\UID=User11\PWD=P1\0DATABASE=DB1\0\0). Список атрибутов завершается двумя null-байтами. При успешном завершении функция возвращает значение TRUE.

## Коды возврата

Все функции ODBC API возвращают значения, называемые кодами возврата. Код возврата определяет, была ли функция выполнена успешно или характеризует тип произошедшей ошибки.

В заголовочном файле *sql.h* определены следующие коды возврата:

```
#define SQL_SUCCESS          0          Функция выполнена
   успешно.

#define SQL_SUCCESS_WITH_INFO 1          Функция выполнена
   успешно, но с
   уведомительным
   сообщением.

#if (ODBCVER >= 0x0300)
#define SQL_NO_DATA          100        Больше нет строк для
   извлечения их из
   результирующего набора. В
   предыдущей версии ODBC
   API этот код возврата
   обозначался как
   SQL_NO_DATA_FOUND. В
   версии 3.x код возврата
   SQL_NO_DATA_FOUND
   содержатся в заголовочном
   файле sqlext.h.
#endif

#define SQL_ERROR            (-1)        При выполнении функции
   произошла ошибка.

#define SQL_INVALID_HANDLE   (-2)        Указан неверный
```

```

#define SQL_STILL_EXECUTING    2    дескриптор.
                                   Функция, выполняемая
                                   асинхронно, пока не
                                   завершена.
#define SQL_NEED_DATA          99   Для успешного выполнения
                                   данной функции следует
                                   предварительно определить
                                   необходимые данные.

```

Первые два кода возврата определяют, что функция была выполнена, а остальные коды возврата информируют о типе произошедшей ошибке.

Для определения типа кода возврата в заголовочном файле `sqltypes.h`

введено следующее объявление:

```
typedef signed short          RETCODE;
```

### Функции ODBC API

В следующей таблице представлен список основных функций ODBC API.

| Функция                               | Описание                                                                                                                                           |
|---------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Соединение с источником данных</b> |                                                                                                                                                    |
| SQLAllocHandle                        | Получает идентификатор (дескриптор) среды, соединения или оператора, или дескриптор приложения.                                                    |
| SQLConnect                            | Соединение с источником данных по DSN, имени и паролю пользователя.                                                                                |
| SQLDriverConnect                      | Соединение с источником данных по указанной строке соединения или при помощи отображаемого диалога для интерактивного ввода параметров соединения. |
| SQLBrowseConnect                      | Последовательно запрашивает атрибуты соединения и устанавливает допустимые значения                                                                |

|                                                                                    |                                                                                                                                    |
|------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------|
|                                                                                    | атрибута. После спецификации значения для каждого требуемого атрибута соединения функция выполняет соединение с источником данных. |
| <b>Получение информации о драйверах и об источниках данных</b>                     |                                                                                                                                    |
| SQLDataSources                                                                     | Возвращает список доступных источников данных.                                                                                     |
| SQLDrivers                                                                         | Возвращает список установленных драйверов и их атрибуты                                                                            |
| SQLGetInfo                                                                         | Возвращает информацию об указанных драйвере и источнике данных.                                                                    |
| SQLGetFunctions                                                                    | Возвращает функции, которые поддерживаются используемым драйвером.                                                                 |
| SQLGetTypeInfo                                                                     | Возвращает информацию о поддерживаемых типах данных.                                                                               |
| <b>Изменение атрибутов драйверов и получение информации об атрибутах драйверов</b> |                                                                                                                                    |
| SQLSetConnectAttr                                                                  | Устанавливает атрибуты соединения.                                                                                                 |
| SQLGetConnectAttr                                                                  | Возвращает значение атрибута соединения.                                                                                           |
| SQLSetEnvAttr                                                                      | Устанавливает атрибуты среды.                                                                                                      |
| SQLGetEnvAttr                                                                      | Возвращает значение атрибута среды.                                                                                                |
| SQLSetStmtAttr                                                                     | Устанавливает атрибуты оператора.                                                                                                  |
| SQLGetStmtAttr                                                                     | Возвращает значение атрибута оператора.                                                                                            |
| <b>Изменение полей дескриптора и получение информации о полях дескриптора</b>      |                                                                                                                                    |
| SQLGetDescField                                                                    | Возвращает значение дескриптора одного поля.                                                                                       |
| SQLGetDescRec                                                                      | Возвращает значения дескриптора для нескольких                                                                                     |

|                                |                                                                                                                                   |
|--------------------------------|-----------------------------------------------------------------------------------------------------------------------------------|
|                                | полей.                                                                                                                            |
| SQLSetDescField                | Устанавливает значение дескриптора для одного поля.                                                                               |
| SQLSetDescRec                  | Устанавливает значение дескриптора для нескольких полей.                                                                          |
| <b>Подготовка SQL-запросов</b> |                                                                                                                                   |
| SQLPrepare                     | Компилирует SQL-оператор для последующего выполнения.                                                                             |
| SQLBindParameter               | Связывает буфер с параметрами, используемыми в SQL-операторе.                                                                     |
| SQLGetCursorName               | Возвращает имя курсора, которое ассоциировано с дескриптором оператора.                                                           |
| SQLSetCursorName               | Определяет имя курсора.                                                                                                           |
| SQLSetScrollOptions            | Устанавливает опции, которые управляют поведением курсора. В версиях ODBC 2.x и 3.x эта функция заменена функцией SQLSetStmtAttr. |
| <b>Выполнение запросов</b>     |                                                                                                                                   |
| SQLExecute                     | Выполняет откомпилированный SQL-оператор.                                                                                         |
| SQLExecDirect                  | Выполняет SQL-оператор.                                                                                                           |
| SQLNativeSql                   | Возвращает текст SQL-оператора, преобразованного конкретным драйвером, но не выполняет его.                                       |
| SQLDescribeParam               | Возвращает описание параметров, используемых в откомпилированном SQL-операторе.                                                   |
| SQLNumParams                   | Возвращает число параметров в откомпилированном SQL-операторе.                                                                    |
| SQLParamData                   | Используется совместно с функцией SQLPutData для передачи во время выполнения значений                                            |



|                                                  |                                                                                                                                                                                                                                                 |
|--------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                                                  | параметров.                                                                                                                                                                                                                                     |
| SQLPutData                                       | Передает часть или все значения параметров.                                                                                                                                                                                                     |
| <b>Извлечение результатов и информации о них</b> |                                                                                                                                                                                                                                                 |
| SQLRowCount                                      | Возвращает число строк, на которые воздействовал SQL-оператор insert, update или delete.                                                                                                                                                        |
| SQLNumResultCols                                 | Возвращает число столбцов в результирующем наборе.                                                                                                                                                                                              |
| SQLDescribeCol                                   | Описывает столбец результирующего набора, возвращая имя поля, его тип, размер и т.п.                                                                                                                                                            |
| SQLColAttribute                                  | Возвращает информацию о столбце результирующего набора. В отличие от функции SQLColAttribute позволяет получить более обширную информацию о столбце, включая информацию, определяемую конкретным драйвером.                                     |
| SQLBindCol                                       | Выполняет связывание буфера приложения-клиента со столбцами результирующего набора.                                                                                                                                                             |
| SQLFetch                                         | Извлекает данные одной следующей строки из результирующего набора, возвращая данные для всех связанных столбцов.                                                                                                                                |
| SQLFetchScroll                                   | Извлекает данные одной или нескольких строк из результирующего набора, возвращая данные для всех связанных столбцов. Функция позволяет явно указать какую строку следует извлечь. Данная функция заменила функцию SQLExtendedFetch из ODBC 2.x. |
| SQLGetData                                       | Извлекает из результирующего набора значение одного столбца одной текущей строки. Для                                                                                                                                                           |

|                         |                                                                                                                                                                              |
|-------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                         | использования этой функции не требуется предварительное связывание столбцов.                                                                                                 |
| SQLSetPos               | Позиционирует курсор в извлеченном блоке данных и позволяет приложению-клиенту: обновлять данные в строке, модифицировать или удалять данные в результирующем наборе.        |
| SQLBulkOperations       | Выполняет несколько вставок или несколько помеченных операций, включая, изменение, удаление и выборку по установленному маркеру.                                             |
| SQLMoreResults          | Определяет, есть ли еще следующий результирующий набор, и при его наличии выполняет переход на него.                                                                         |
| SQLGetDiagField         | Возвращает значение поля записи из структуры диагностической информации, ассоциированной с конкретным дескриптором (среды, соединения, оператора).                           |
| SQLGetDiagRec           | Возвращает значения нескольких predetermined полей записи из структуры диагностической информации, ассоциированной с конкретным дескриптором (среды, соединения, оператора). |
| <b>Функции каталога</b> |                                                                                                                                                                              |
| SQLColumnPrivileges     | Возвращает список полей и имеющиеся привилегии для одной или нескольких таблиц.                                                                                              |
| SQLColumns              | Возвращает список имен полей в указанной таблице.                                                                                                                            |

|                               |                                                                                                                                                                                                |
|-------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| SQLForeignKeys                | Возвращает список полей, которые составляют внешние ключи таблицы, если они созданы.                                                                                                           |
| SQLPrimaryKeys                | Возвращает список полей, которые составляют первичный ключ таблицы.                                                                                                                            |
| SQLProcedureColumns           | Возвращает в виде результирующего набора список входных и выходных параметров указанной процедуры.                                                                                             |
| SQLProcedures                 | Возвращает список хранимых процедур для подключенного источника данных.                                                                                                                        |
| SQLSpecialColumns             | Получает информацию об оптимальном наборе полей уникально идентифицирующих строку в указанной таблице или имя поля, которое автоматически обновляется при изменении какого-либо поля в строке. |
| SQLStatistics                 | Возвращает информацию о таблице и список индексов, ассоциированных с ней.                                                                                                                      |
| SQLTablePrivileges            | Возвращает в виде результирующего набора список таблиц и привилегии, назначенные для каждой таблицы.                                                                                           |
| SQLTables                     | Возвращает в виде результирующего набора список таблиц, хранимых в указанном источнике данных.                                                                                                 |
| <b>Освобождение оператора</b> |                                                                                                                                                                                                |
| SQLFreeStmt                   | Завершает обработку оператора, удаляет результирующий набор и освобождает все ресурсы, ассоциированные с данным дескриптором оператора.                                                        |
| SQLCloseCursor                | Закрывает курсор, открытый с данным дескриптором оператора, и удаляет                                                                                                                          |

|                                |                                                                                                                                                                                                                                                                                                        |
|--------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                                | резльтирующий набор.                                                                                                                                                                                                                                                                                   |
| SQLCancel                      | Завершает выполнение SQL-оператора, прекращая асинхронное выполнение функции, выполнение функции, требующей данные или функции, выполняемой в другом потоке. В отличие от версии 2.x данная функция не может выполнить освобождение дескриптора оператора и требуется явный вызов функции SQLFreeStmt. |
| SQLEndTran                     | Выполняет завершение или откат транзакции.                                                                                                                                                                                                                                                             |
| <b>Освобождение соединения</b> |                                                                                                                                                                                                                                                                                                        |
| SQLDisconnect                  | Закрывает соединение с источником данных.                                                                                                                                                                                                                                                              |
| SQLFreeHandle                  | Освобождает ресурсы, ассоциированные с указанным дескриптором (среды, соединения, оператора, приложения).                                                                                                                                                                                              |

## Дескрипторы

Перед использованием функций ODBC API приложение-клиент создает *дескриптор (идентификатор) окружения*, определяющий глобальный контекст для доступа к источникам данных. Дескриптор окружения предоставляет доступ к различной информации, включая текущие установки всех атрибутов окружения, дескрипторы соединений, созданные для данного окружения, диагностику уровня окружения.

Дескриптор окружения определяет некоторую структуру, содержащую данную информацию. Непосредственно дескриптор окружения обычно используется при вызове функций SQLDataSources и SQLDrivers и при создании дескрипторов соединения.

Для приложения-клиента, реализующего с использованием функций ODBC API доступ к источнику данных, достаточно иметь один дескриптор окружения.

Создание дескриптора окружение выполняется функцией `SQLAllocHandle`, а освобождение – функцией `SQLFreeHandle`.

Функция `SQLAllocHandle` введена в версии ODBC 3.x вместо существовавших в версии ODBC 2.0 функций `SQLAllocConnect`, `SQLAllocEnv` и `SQLAllocStmt`. Для того чтобы приложение, использующее функцию `SQLAllocHandle`, могло работать через драйверы ODBC 2.x, менеджер драйверов версии 3.x заменяет вызовы функций третьей версии на их аналогичные второй версии и передает такой "откорректированный" вызов ODBC-драйверу.

Для подключения к базе данных следует создать *дескриптор (идентификатор) соединения*. Для одного дескриптора окружения может быть создано несколько дескрипторов соединения.

Для выполнения SQL-оператора создается дескриптор (идентификатор) оператора.

Для одного дескриптора соединения может быть создано несколько дескрипторов оператора.

По спецификации ODBC для каждого приложения драйверы могут поддерживать неограниченное число дескрипторов каждого типа. Однако конкретный драйвер может накладывать некоторые ограничения на количество дескрипторов.

Функция, используемая для создания дескриптора окружения, соединения, оператора или приложения имеет следующее формальное описание:

```
SQLRETURN SQLAllocHandle (  
    SQLSMALLINT      HandleType,  
    SQLHANDLE         InputHandle,  
    SQLHANDLE *       OutputHandlePtr);
```

Параметр *HandleType* ([Input]) указывает одной из следующих констант тип создаваемого дескриптора:

```
SQL_HANDLE_ENV  
SQL_HANDLE_DBC
```

SQL\_HANDLE\_STMT  
SQL\_HANDLE\_DESC

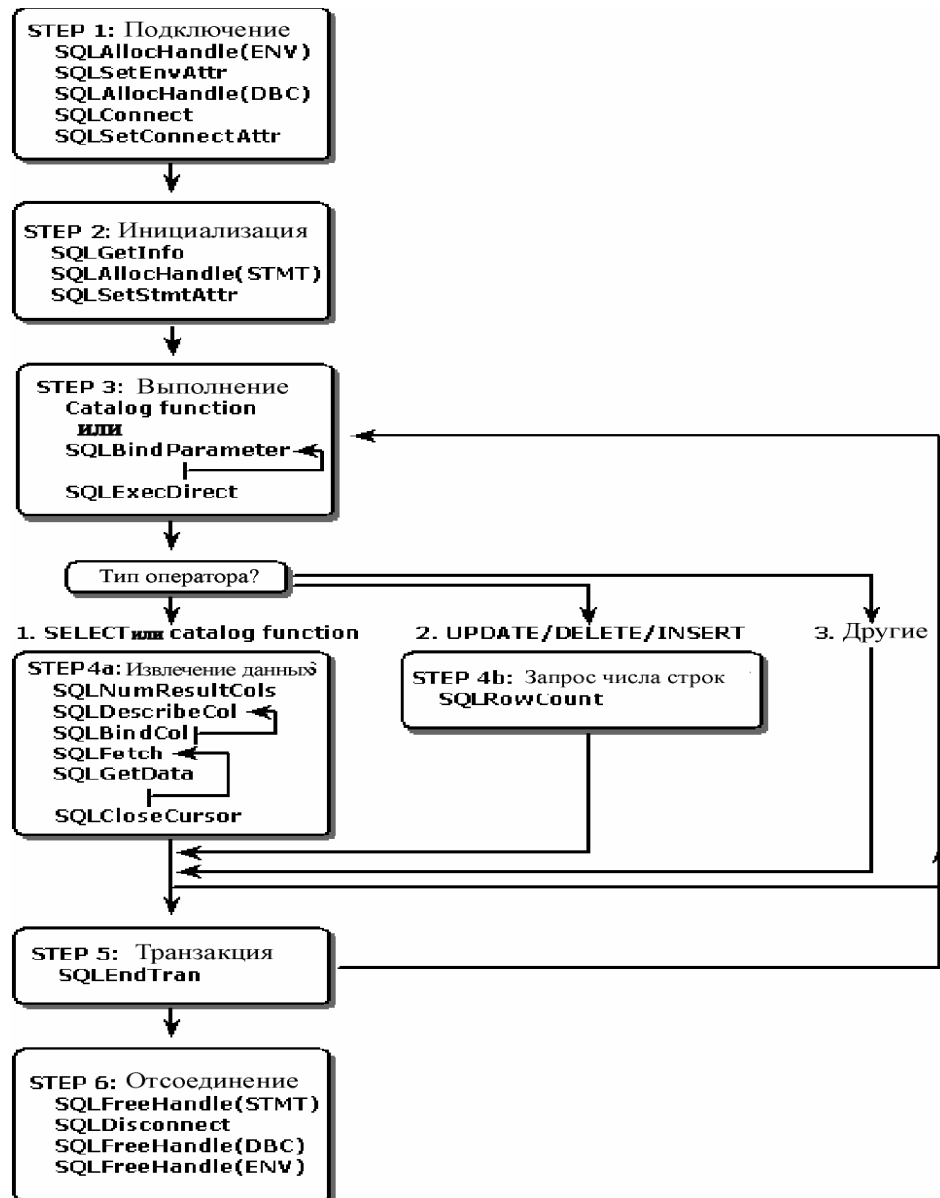
Параметр *InputHandle* ([Input]) определяет контекст, в который добавляется создаваемый дескриптор. Если тип дескриптора SQL\_HANDLE\_ENV, то параметр *InputHandle* указывается константой SQL\_NULL\_HANDLE. При создании дескриптора среды параметр *InputHandle* задает дескриптор окружения, а для создания дескриптора оператора (SQL\_HANDLE\_STMT) и дескриптора приложения (SQL\_HANDLE\_DESC) – дескриптор соединения. Идентификаторы, определяющие тип дескриптора и сам дескриптор описаны в заголовочных файлах sql.h и sqltypes.h.

Параметр *OutputHandlePtr* ([Output]) –это указатель на буфер, в который помещается создаваемая для дескриптора структура данных.

После создания дескриптора окружения следует установить атрибут SQL\_ATTR\_ODBC\_VERSION. В противном случае при попытке создать дескриптор соединения произойдет ошибка.

### **Схема подключения к источнику данных**

На следующей схеме отображена последовательность действий приложения-клиента для реализации доступа к источнику данных.



## **Соединение с источником данных**

Для непосредственного подключения к базе данных ODBC API предоставляет следующие три функции:

- SQLConnect - соединение с источником данных по DSN, имени и паролю пользователя;
- SQLDriverConnect - соединение с источником данных по указанной строке соединения или при помощи отображаемого диалога для интерактивного ввода параметров соединения;
- SQLBrowseConnect - соединение с источником данных с предварительным последовательным запросом атрибутов соединения.

## **Пул соединений**

Организация пула соединений позволяет приложению выбирать соединения из пула без необходимости переустанавливать их для каждого использования. После того как соединение создано и помещено в пул приложение может использовать это соединение, не повторяя опять процесс установки соединения.

Использование соединений, помещенных в пул, может значительно увеличить производительность приложений, многократно устанавливающих и разрывающих соединения. Примером таких приложений могут служить серверные Интернет-приложения среднего звена в трехзвенной архитектуре, постоянно повторно устанавливающие и разрывающие соединения.

## **Выборка данных**

Для извлечения данных с использованием ODBC API сначала следует вызвать функцию, выполняющую SQL-оператор, который определяет формируемый результирующий набор. И только затем можно приступить к выборке данных.

ODBC API предоставляет два способа извлечения данных из результирующего набора:



- с предварительным связыванием полей результирующего набора с переменными основного языка программирования;
- прямая выборка каждого поля результирующего набора в указываемую переменную основного языка программирования.

Результирующий набор создается при выполнении SQL-оператора SELECT. Для выполнения запроса ODBC API предоставляет следующие две функции:

- `SQLExecute` - выполняет откомпилированный SQL-оператор;
- `SQLExecDirect` - выполняет SQL-оператор, указываемый параметром.

Функция `SQLExecDirect` реализует *одношаговый интерфейс*, при котором процесс компиляции SQL-оператора и его выполнение осуществляется единожды при вызове данной функции.

Функции `SQLPrepare` и `SQLExecute` реализуют *многошаговый интерфейс*: сначала выполняется компиляция оператора и формируется план выполнения, а затем возможно многократное выполнение подготовленного оператора.

Функция `SQLExecDirect` имеет следующее формальное описание:

```
SQLRETURN SQLExecDirect (  
    SQLHSTMT      StatementHandle,  
    SQLCHAR *      StatementText,  
    SQLINTEGER     TextLength) ;
```

Параметр *StatementHandle* ([Input]) указывает дескриптор оператора, параметр *StatementText* ([Input]) определяет выполняемый SQL-оператор, *TextLength* (Input) - длина строки \**StatementText*.

При выполнении SQL-оператора, содержащего параметры, их значения предварительно должны быть определены вызовом функций `SQLParamData` и `SQLPutData` или `SQLBindParameter`. В противном случае при выполнении функции `SQLExecDirect` будет возвращено значение `SQL_NO_DATA`.

При многошаговом интерфейсе первой должна быть выполнена функция `SQLPrepare`, иницилирующая компиляцию SQL-оператора.

Функция `SQLPrepare` имеет следующее формальное описание:

```
SQLRETURN SQLPrepare (
    SQLHSTMT      StatementHandle,
    SQLCHAR *      StatementText,
    SQLINTEGER     TextLength);
```

Параметр StatementHandle ([Input]) указывает дескриптор оператора, параметр StatementText ([Input]) определяет текст компилируемого SQL-оператора, TextLength ([Input]) – это длина строки \*StatementText.

Функция SQLExecute выполняет откомпилированный SQL-оператор. Если выполняемым SQL-оператором был оператор SELECT, то в результате выполнения SQLExecute создается результирующий набор.

Функция SQLExecute имеет следующее формальное описание:

```
SQLRETURN SQLExecute (
    SQLHSTMT      StatementHandle);
```

Параметр StatementHandle ([Input]) указывает дескриптор оператора

Например:

```
SQLHSTMT hstmtS, hstmtU;
SQLExecDirect(hstmtS, "SELECT F2 FROM TBL1 ", SQL_NTS);
SQLPrepare(hstmtU,
    "UPDATE TBL1 SET F2=F2*10 WHERE F1=20", SQL_NTS);
SQLExecute(hstmtU);
```

Результирующий набор формируется в выделяемой области памяти. Для того чтобы использовать данные, записанные в результирующий набор, их следует извлечь из той области памяти в переменные используемого языка программирования.

Извлечение данных из результирующего набора в переменные может быть выполнено:

- вызовом функции SQLFetch или SQLFetchScroll;
- вызовом функции SQLGetData.

Функции SQLFetch или SQLFetchScroll всегда выполняют продвижение курсора на следующую запись. При этом функция SQLFetch, реализует механизм "однонаправленного курсора", а функция SQLFetchScroll, позволяет в зависимости от используемого источника данных реализовывать механизм "двунаправленного курсора" и механизм "прямой выборки".

Функции SQLFetch или SQLFetchScroll выполняют одновременное извлечение данных только в том случае, если поля результирующего набора предварительно были связаны с переменными вызовом функции SQLBindCol для каждого связываемого поля.

Функция SQLFetch имеет следующее формальное описание:

```
SQLRETURN SQLFetch (  
    SQLHSTMT      StatementHandle);
```

Параметр StatementHandle ([Input]) указывает дескриптор оператора

Функция SQLGetData имеет следующее формальное описание:

```
SQLRETURN SQLGetData (  
    SQLHSTMT      StatementHandle,  
    SQLUSMALLINT   ColumnNumber,  
    SQLSMALLINT    TargetType,  
    SQLPOINTER     TargetValuePtr,  
    SQLINTEGER     BufferLength,  
    SQLINTEGER *   StrLen_or_IndPtr);
```

Параметр StatementHandle ([Input]) указывает дескриптор оператора

Параметр ColumnNumber ([Input]) указывает номер связываемого столбца результирующего набора (начиная с 1). По умолчанию столбец номер 0 является столбцом маркера строки, в том случае если маркеры доступны.

Параметр TargetType ([Input]) определяет C-тип данных для буфера\*TargetValuePtr. Параметр TargetValuePtr ([Output]) определяет буфер, в который выполняется извлечение данных, а параметр BufferLength ([Input]) определяет размер этого буфера в байтах. Для данных, имеющих фиксированную длину, таких как целочисленные значения, драйвер игнорирует значение параметра BufferLength.

Параметр StrLen\_or\_IndPtr ([Output]) определяет буфер, в котором возвращается размер данных или индикатор

Следующий пример иллюстрирует применение механизма построчной выборки данных без их предварительного связывания.

```
//ODBC_Connect.cpp
#include "stdafx.h"
#include "ODBC_connect.h"
#include <iostream>

#ifdef _DEBUG
#define new DEBUG_NEW
#endif

CWinApp theApp; // Объявление объекта приложения
using namespace std;
int _tmain(int argc, TCHAR* argv[], TCHAR* envp[])
{
    int nRetCode = 0;
    if (!AfxWinInit(::GetModuleHandle(NULL), NULL,
                   ::GetCommandLine(), 0))
    {
        _tprintf(_T("Ошибка инициализации MFC\n"));
        nRetCode = 1;
    }
    else
    {
        std::cout<<"Begin"<<std::endl;
        SQLHENV    henv;      // Дескриптор окружения
        SQLHDBC    hdbc;      // Дескриптор соединения
        SQLHSTMT    hstmt;     // Дескриптор оператора
        SQLRETURN    retcode;   // Код возврата
        /*Инициализация дескриптора окружения */
    }
}
```



```
        if (retcode == SQL_ERROR || retcode == SQL_SUCCESS_WITH_INFO)
        {
            }
            if (retcode == SQL_SUCCESS || retcode ==
SQL_SUCCESS_WITH_INFO) {
                /* Извлечение данных трех полей результирующего набора */
                SQLGetData(hstmt, 1, SQL_C_ULONG, &sqf1, 0, &sbfl);
                SQLGetData(hstmt, 2, SQL_C_ULONG, &sqf2, 0, &sbfl);
                SQLGetData(hstmt, 3, SQL_C_CHAR, sqf3, 50, &sbfl);
                /* Запись в поток вывода строк результирующего набора */
std::cout<< "1: "<<sqf1<<" 2:  "<<sqf2<<" 3:  "<< sqf3<<"
"<<std::endl;
            } else {
                break;
            }
        }
    }

    /* Освобождение дескрипторов */
    if (retcode == SQL_SUCCESS || retcode ==
SQL_SUCCESS_WITH_INFO) {
        SQLFreeHandle(SQL_HANDLE_STMT, hstmt);
    }
    SQLDisconnect(hdbc);
}
SQLFreeHandle(SQL_HANDLE_DBC, hdbc);
}
SQLFreeHandle(SQL_HANDLE_ENV, henv);
}
return nRetCode;
```

```

}
}

//ODBC_Connect.h
#pragma once
#include "resource.h"
#ifdef _AFX_NOFORCE_LIBS
////////////////////////////////////
// Win32 библиотеки
#pragma comment(lib, "odbc32.lib")
#pragma comment(lib, "odbccp32.lib")
#endif // !_AFX_NOFORCE_LIBS
#ifdef __SQL
    #include <sql.h>           // ядро
#endif
#ifdef __SQULEXT
    #include <sqlext.h>       // расширение
#endif

```

Функция, выполняющая предварительное связывание данных имеет следующее формальное описание:

```

SQLRETURN SQLBindCol(
    SQLHSTMT      StatementHandle,
    SQLUSMALLINT   ColumnNumber,
    SQLSMALLINT    TargetType,
    SQLPOINTER     TargetValuePtr,
    SQLINTEGER     BufferLength,
    SQLLEN *       StrLen_or_Ind);

```

Значения параметров функции `SQLBindCol` аналогичны значениям параметров функции `SQLGetData`. Но функция `SQLBindCol` указывается только один раз для каждого поля, а затем выборка данных выполняется автоматически при вызове функции `SQLFetch`.

## Параметрические запросы

Параметром называется переменная, используемая в SQL-операторе.

Применение параметров позволяет формировать SQL-операторы непосредственно во время выполнения приложения. Так, при использовании многошагового интерфейса, при котором процесс компиляции и выполнения SQL-оператора происходит как последовательные действия, достаточно откомпилировать SQL-оператор с параметрами вместо явного указания значений полей таблицы базы данных, а затем многократно выполнять данный оператор с различными значениями параметров.

Например:

```
INSERT INTO TBL1 (F_ID, F2, F3) VALUES (?, ?, ?)
```

Параметры могут быть *именованными* и *позиционными*.

Позиционные параметры указываются символом вопросительный знак (?), называемым маркером параметров. При выполнении оператора вместо параметра в соответствующую позицию SQL-оператора приставляется значение параметра.

Выполняемый SQL-оператор может содержать несколько параметров.

Перед использованием параметр следует определить (выполнить связывание параметра). Определение параметра заключается в указании типов для значения (C-тип) и для поля таблицы (SQL-тип), а также указание буфера, в котором будет содержаться значение параметра, или номера параметра - для его последующего запроса во время выполнения. Определение параметра реализуется функцией `SQLBindParameter`.

Функция `SQLBindParameter` имеет следующее формальное описание:

```
SQLRETURN SQLBindParameter(  
    SQLHSTMT      StatementHandle,
```



```

SQLUSMALLINT      ParameterNumber,
SQLSMALLINT       InputOutputType,
SQLSMALLINT       ValueType,
SQLSMALLINT       ParameterType,
SQLUINTEGER       ColumnSize,
SQLSMALLINT       DecimalDigits,
SQLPOINTER        ParameterValuePtr,
SQLINTEGER        BufferLength,
SQLINTEGER *      StrLen_or_IndPtr);

```

Параметр *StatementHandle* ([Input]) указывает дескриптор оператора, параметр *ParameterNumber* ([Input]) задает номер параметра (по мере их вхождения в SQL-оператор), начиная с 1.

Параметр *InputOutputType* ([Input]) определяет тип параметра.

Параметр *ValueType* ([Input]) определяет тип значения переменной (C-тип), из которой будет извлекаться значение, передаваемое в базу данных, а параметр *ParameterType* ([Input]) указывает тип параметра (SQL-тип поля таблицы базы данных).

Параметр *ColumnSize* ([Input]) определяет размер столбца или указывает выражение соответствующее маркеру параметра, а параметр *DecimalDigits* ([Input]) определяет количество десятичных знаков в столбце или указывает выражение соответствующее маркеру параметра. Получить размер столбца и количество десятичных знаков в столбце можно при помощи функции ODBC API *SQLDescribeCol*.

Параметр *ParameterValuePtr* ([Deferred Input]) является указателем на буфер для данных, передаваемых в качестве параметра. Длина этого буфера определяется параметром *BufferLength* ([Input/Output]), а указатель на буфер для длины параметра задается параметром *StrLen\_or\_IndPtr* ([Deferred Input]).

Если перед выполнением функции *SQLExecDirect*, значения параметров, используемых в запросе не переданы на сервер, то функции возвращает код

ответа SQL\_NEED\_DATA. Для передачи параметров в приложении могут использоваться функции SQLParamData и SQLPutData. Функция SQLParamData используется совместно с функцией SQLPutData для передачи значений параметров во время выполнения. Если функции SQLParamData возвращает значение SQL\_NEED\_DATA, то она также возвращает и номер параметра, для которого следует ввести значение. Передача значения параметра выполняется функцией SQLPutData.

Именованные параметры могут использоваться в том случае, когда в SQL-операторе выполняется вызов хранимой процедуры. Именованные параметры идентифицируются в соответствии с их именами, а не по порядку их расположения (как при позиционных параметрах). Именованные параметры, также как и позиционные параметры связываются с переменной посредством вызова функции ODBC API SQLBindParameter, но идентифицируются посредством поля SQL\_DESC\_NAME IPD-дескриптора (Implementation Parameter Descriptor). Именованные параметры также могут быть связаны с переменной при вызове функции SQLSetDescField или функции SQLSetDescRec.

Далее приведен пример использования именованных параметров, используемых при вызове предварительно созданной хранимой процедуры t1 с двумя параметрами

```
(CREATE PROCEDURE t1 @f_id int = 10, @f2 char(30) AS ).
```

Первый параметр процедуры имеет значение по умолчанию равное 10, а второй параметр @f2 обязательно должен быть указан при вызове процедуры. Параметр @f2 это динамический параметр, называемый именованным параметром.

```
// Компилирование вызова хранимой процедуры
SQLPrepare(hstmt, "{call t1(?)}", SQL_NTS);
// Заполнение записи 1 для IPD-дескриптора
SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT,
                  SQL_C_CHAR, SQL_CHAR,
```

```
        30, 0, sz_Fl2, 0, &cbValue);  
// Получение IPD-дескриптора и определение полей  
// SQL_DESC_NAMED и SQL_DESC_UNNAMED для записи #1.  
SQLGetStmtAttr(hstmt, SQL_ATTR_IMP_PARAM_DESC,  
               &hIpd, 0, 0);  
SQLSetDescField(hIpd, 1, SQL_DESC_NAME,  
               "@f12", SQL_NTS);  
SQLSetDescField(hIpd, 1, SQL_DESC_UNNAMED,  
               SQL_NAMED, 0);  
// Если переменная sz_Fl2 была корректно  
// инициализирована, то можно выполнять  
// вызов хранимой процедуры  
SQLExecute(hstmt);
```

## Курсоры

Применение курсоров позволяет приложению выполнять выборку одной или нескольких строк за одну операцию извлечения данных. Также курсоры поддерживают возможность применения операторов UPDATE, INSERT и DELETE к текущей позиции курсора.

Для использования курсора следует установить соединение с базой данных и установить нужные значения атрибутам оператора, контролирующим поведение курсора.

Для использования курсора требуется выполнить следующие действия:

1. Установить атрибуты курсора, вызвав функцию SQLSetStmtAttr. Эта функция позволяет устанавливать следующие атрибуты :  
SQL\_ATTR\_CURSOR\_TYPE и SQL\_ATTR\_CONCURRENCY,  
или  
SQL\_CURSOR\_SCROLLABLE и SQL\_CURSOR\_SENSITIVITY..
2. Определите размер результирующего набора, вызвав функцию SQLSetStmtAttr с атрибутом SQL\_ATTR\_ROW\_ARRAY\_SIZE.

3. Для того чтобы использовать позиционированный SQL-оператор с фразой `WHERE CURRENT OF` следует определить имя курсора, вызвав функцию `SQLSetCursorName`.
4. Для создания результирующего набора следует выполнить SQL-оператор `SELECT`, вызвав функцию `SQLExecute` или функцию `SQLExecDirect`.
5. Если имя курсора не было предварительно определено, но требуется использовать возможность применения позиционированного SQL-оператора с фразой `WHERE CURRENT OF`, то следует создать имя курсора, вызвав функцию `SQLGetCursorName`.
6. Для получения информации о количестве столбцов в сформированном результирующем наборе можно вызвать функцию `SQLNumResultCols`.
7. Определите связывание столбцов результирующего набора с буфером, предназначенном для извлечения значения столбца (одной строки или сразу набора строк).
8. Выполните функцию `ODBC API`, выполняющую извлечение строк из результирующего набора в связанный буфер (или функцию выполняющую извлечение поочередно каждого столбца в указываемый параметром буфер, если не было выполнено предварительного связывания).
9. Если выполняемый SQL-оператор состоял из нескольких операторов `SELECT`, то сформированный результирующий набор будет состоять из нескольких множеств. Для перехода к следующему множеству (результирующему набору) используется функция `SQLMoreResults`. В том случае, если следующее множество существует, то эта функция выполняет переход к нему и возвращает код ответа `SQL_SUCCESS`. Если выбраны все множества сформированного результирующего набора, то функция возвращает код ответа `SQL_NO_DATA`.
10. Для освобождения дескриптора оператора вызовите функцию `SQLFreeStmt`. При этом для того, чтобы одновременно выполнить освобождение связанных с данным оператором буферов,

используемых для извлечения значений столбцов, установите значение параметра *fOption* равным SQL\_UNBIND.

Приложение, использующее функции ODBC API управляет поведением курсора, устанавливая атрибуты оператора. Можно использовать два различных способа определения характеристик курсора:

- определить *тип курсора*;
- определить *поведение курсора*.

Для определения типа курсора вызывается функция SQLSetStmtAttr со значением атрибута SQL\_ATTR\_CURSOR\_TYPE, который имеет тип SQLINTEGER и может задаваться следующими значениями:

- SQL\_CURSOR\_FORWARD\_ONLY – однонаправленный курсор;
- SQL\_CURSOR\_STATIC статический курсор, определяющий что информация, извлеченная в результирующий набор не будет отражать изменение данных в БД, произошедшие после создания результирующего набора;
- SQL\_CURSOR\_KEYSET\_DRIVEN – курсор, управляемый ключом. Такой курсор позволяет "видеть" изменение и удаление строк в БД, произошедшие после создания результирующего набора, но не отображает создание новых строк. Количество строк, для которых создаются ключи указывается атрибутом SQL\_ATTR\_KEYSET\_SIZE;
- SQL\_CURSOR\_DYNAMIC – динамический курсор, отражающий изменение данных в БД после создания результирующего набора.

На используемый тип курсора накладывает ограничение применяемый ODBC-драйвер. Так, многие драйверы не поддерживают возможность применения динамического курсора

Для определения поведения курсора вызывается функция SQLSetStmtAttr со значениями атрибутов SQL\_ATTR\_CURSOR\_SCROLLABLE и SQL\_ATTR\_CURSOR\_SENSITIVITY

Атрибут SQL\_ATTR\_CURSOR\_SCROLLABLE управляет *перемещаемым курсором*, имеет тип `SQLINTEGER` и может задаваться следующими двумя значениями:

- `SQL_NONSCROLLABLE` – перемещаемый курсор реализуется как простой однонаправленный курсор. При вызове в приложении функции `SQLFetchScroll` параметр *FetchOrientation* может принимать только значение `SQL_FETCH_NEXT`.
- `SQL_SCROLLABLE` – перемещаемый курсор, поддерживающий двунаправленный просмотр результирующего набора, а также прямую выборку строк.

Перемещаемые курсоры для извлечения данных используют функцию `SQLFetchScroll`. Если простой однонаправленный курсор, используемый функцией `SQLFetch`, позволяет перемещаться только в одном прямом направлении и выполнять выборку только одной строки за один вызов функции, то перемещаемый курсор позволяет:

- выполнять за один вызов функции извлечение более одной строки;
- перемещаться в двух направлениях и на любое число строк;
- выполнять прямую выборку строки из результирующего набора по ее номеру.

Атрибут `SQL_ATTR_CURSOR_SENSITIVITY` определяет чувствительность курсора к изменениям, выполняемым другими курсорами, имеет тип `SQLINTEGER` и может определять режимы, задаваемые следующими значениями:

- `SQL_UNSPECIFIED` – неопределенный курсор, при котором изменения могут быть сделаны как видимыми, так и невидимыми, или частично видимыми. Этот режим используется по умолчанию;
- `SQL_INSENSITIVE` – режим нечувствительного курсора, при котором все курсоры показывают результирующий набор без отражения изменений, выполненных для других курсоров. Нечувствительный курсор является курсором "только чтение". Он

соответствует статическому типу курсора с уровнем изоляции "только чтение";

- SQL\_SENSITIVE – режим чувствительного курсора, при котором курсоры "видят" все изменения, выполненные другими курсорами.

Функция `SQLSetStmtAttr` позволяет установить значения атрибутов оператора и имеет следующее формальное описание:

```
SQLRETURN SQLSetStmtAttr (  
    SQLHSTMT      StatementHandle,  
    SQLINTEGER     Attribute,  
    SQLPOINTER     ValuePtr,  
    SQLINTEGER     StringLength);
```

Параметр *StatementHandle* ([Input]) указывает дескриптор оператора.

Параметр *Attribute* ([Input]) определяет атрибут, значение которого устанавливается, а параметр *ValuePtr* ([Input]) является указателем на значение, назначаемое атрибуту *Attribute*.

Параметр *StringLength* ([Input]) зависит от типа значения, передаваемого параметром *ValuePtr* и от вида атрибута, и может содержать длину строки, игнорироваться или указываться константами такими, как `SQL_NTS`, `SQL_IS_POINTER`.

## Именованные курсоры

Курсор, для которого определено имя курсора, называется *именованным курсором*.

Ассоциировать имя курсора с активным дескриптором оператора можно вызовом функции `SQLSetCursorName`. В том случае если эта функция не вызывается явным образом, то драйвер при необходимости может генерировать имя курсора при выполнении SQL-оператора.

Следующий пример иллюстрирует применение функции `SQLSetCursorName` для получения имени курсора и выполнения затем позиционированного SQL-оператора `UPDATE`.

```
#define N_LEN 24
#define PH_LEN 7

SQLHSTMT      hstmtS;  // Дескриптор оператора
                  // (для оператора SELECT)
SQLHSTMT      hstmtU;  // Дескриптор оператора
                  // (для оператора UPDATE)

SQLRETURN      retcode;
SQLHDBC        hdbc;
SQLCHAR        szN[N_LEN], szPh[PH_LEN];
SQLINTEGER      cbN, cbPh;

/* Создаем дескриптор оператора и определяем имя
курсора */
SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmtS);
SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmtU);
SQLSetCursorName(hstmtS, "C1", SQL_NTS);

/* Выполняем SQL-оператор SELECT для формирования
результатирующего набора и связываем его столбцы с
локальными буферами для извлечения данных */
SQLExecDirect(hstmtS,
               "SELECT N, PH FROM Tbl1", SQL_NTS);
SQLBindCol(hstmtS, 1, SQL_C_CHAR,
            szN, N_LEN, &cbN);
SQLBindCol(hstmtS, 2, SQL_C_CHAR,
            szPh, PH_LEN, &cbPh);
```



```
/* Выбираем строки результирующего набора до тех пор,  
пока не найдем строку "Антипов А." */  
do  
    retcode = SQLFetch(hstmtS);  
while ((retcode == SQL_SUCCESS || retcode ==  
SQL_SUCCESS_WITH_INFO) &&  
    (strcmp(szN, "Антипов А.") != 0));  
  
/* Выполняем позиционированный UPDATE для текущей  
выбранной строки именованного курсора */  
if (retcode == SQL_SUCCESS || retcode ==  
SQL_SUCCESS_WITH_INFO) {  
    SQLExecDirect(hstmtU,  
        "UPDATE Tbl2 SET PH=\"1373322\"  
        WHERE CURRENT OF C1", SQL_NTS);  
}
```

### **Блочная выборка данных**

При использовании перемещаемого курсора для изменения текущей позиции курсора и выборки строк используется функция **SQLFetchScroll**. Эта функция позволяет реализовывать:

- *относительный скроллинг* - перемещение по результирующему набору в двух направлениях и на любое число строк;
- *абсолютный скроллинг* – перемещение на первую или последнюю строку, или строку с указанным номером.

Функция **SQLFetchScroll** выполняет выборку *набора строк* из сформированного *результатирующего набора* и возвращает данные для всех связанных столбцов. Наборы строк (rowset) могут быть указаны как через абсолютное или относительное позиционирование, так и посредством

закладок(bookmark). В версии ODBC 2.x для этих целей использовалась функция `SQLExtendedFetch`.

Функция `SQLFetchScroll` имеет следующее формальное описание:

```
SQLRETURN SQLFetchScroll (  
    SQLHSTMT      StatementHandle,  
    SQLSMALLINT    FetchOrientation,  
    SQLINTEGER     FetchOffset) ;
```

Параметр *StatementHandle* ([Input]) указывает дескриптор оператора.

Перемещение курсора определяется типом выборки, указывается параметром *FetchOrientation* ([Input]) и может принимать следующие значения:

- `SQL_FETCH_NEXT` – переход к следующей строке (относительный скроллинг);
- `SQL_FETCH_PRIOR` – переход к предыдущей строке (относительный скроллинг);
- `SQL_FETCH_FIRST` – переход к первой строке (абсолютный скроллинг);
- `SQL_FETCH_LAST` – переход к последней строке (абсолютный скроллинг);
- `SQL_FETCH_ABSOLUTE` – переход к строке с указанным номером (абсолютный скроллинг);
- `SQL_FETCH_RELATIVE` – перемещение вперед или назад на указанное количество строке (относительный скроллинг);
- `SQL_FETCH_BOOKMARK` – переход к строке по закладке (абсолютный скроллинг).

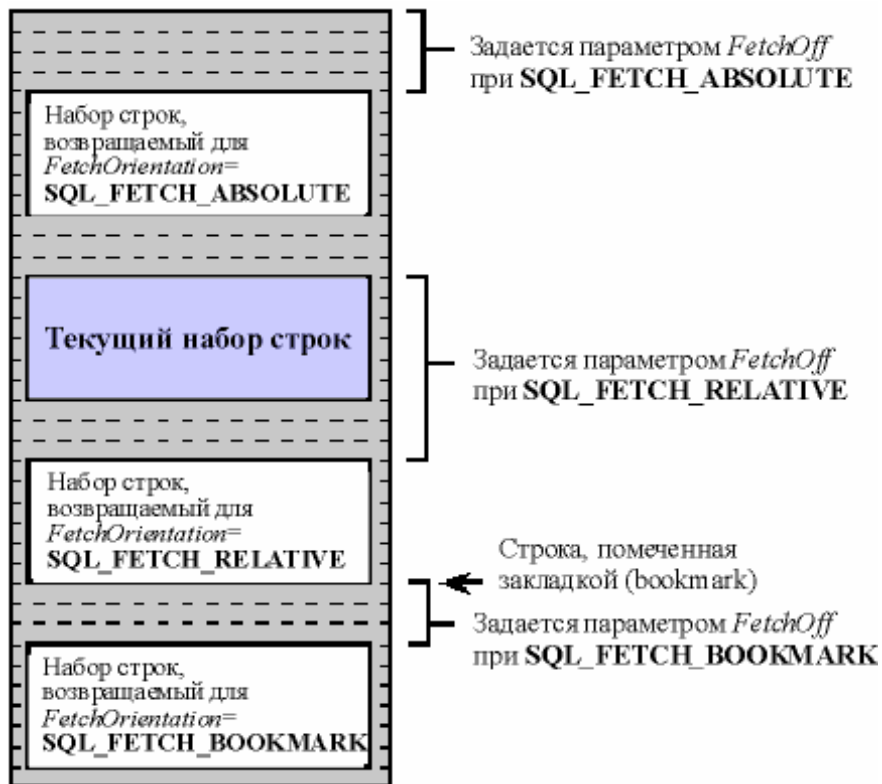
Количество строк, на которые выполняется перемещение курсора и номер абсолютной позиции, указывается параметром *FetchOffset* ([Input]).

Параметры *FetchOrientation* и *FetchOffset* функции `SQLFetchScroll` совместно определяют набор строк, который будет извлечен из результирующего набора.

На следующей схеме показан механизм блочной выборки строк при позиционировании на следующую, предыдущую, первую или последнюю строку.



Механизм выборки строк функцией *SQLFetchScroll* также позволяет реализовывать позиционирование по абсолютному указанному номеру, позиционирование со смещением на заданное число строк. или позиционирование посредством закладки. Эти механизмы отображены на следующей схеме.



Функция `SQLFetchScroll` выполняет позиционирование курсора на указанную строку результирующего набора и возвращает набор строк, начиная с установленной позиции курсора.

После того, как набор строк извлечен, для позиционирования в нем конкретной строки или извлечения строки можно использовать функцию `SQLSetPos`.

При использовании блочного курсора за один вызов функции может возвращаться несколько строк. Поэтому с каждым столбцом должна

связываться не просто переменная соответствующего типа, а массив. Такой массив обычно называется *буфером набора строк* (rowset buffer).

Возможно два типа связывания:

- *связывание по столбцу* (column-wise binding), при котором с каждым столбцом связывается отдельный массив (структура данных, содержащая элементы одного типа);
- *связывание по строке* (row-wise binding), при котором с каждой строкой связывается отдельный массив (структура данных, типы элементов которой соответствуют типам столбцов строки набора данных).

Для выполнения любого типа связывания используется функция ODBC API SQLBindCol. При этом тип связывания определяется атрибутом SQL\_ATTR\_ROW\_BIND\_TYPE. В случае использования блочного курсора со связыванием по строке или по столбцу функции SQLBindCol в качестве параметра вместо адреса простой переменной передается адрес массива.

При использовании связывания по столбцу с каждым столбцом может быть связано от одного до трех массивов: первый массив – для извлекаемых значений, второй – для длины/ индикатора буферов, третий – для индикатора буферов (если индикаторы и длина хранятся по отдельности). Каждый массив должен содержать число элементов равное числу строк в извлекаемом наборе строк.

Следующий пример иллюстрирует применение связывания по столбцам для набора строк, состоящего из одного столбца.

```
#define ROW_ARRAY_SIZE 30          // Кол-во строк
                                   // в наборе строк

SQLINTEGER    IDlArray[ROW_ARRAY_SIZE],
               NumRowsFetched;
SQLINTEGER    IDlIndArray[ROW_ARRAY_SIZE];
SQLUSMALLINT  RowStatusArray[ROW_ARRAY_SIZE], i;
```

```
SQLRETURN      rc;
SQLHSTMT       hstmt;

// Устанавливаем атрибут оператора // SQL_ATTR_ROW_BIND_TYPE для
// использования связывания
// по столбцам
SQLSetStmtAttr(hstmt, SQL_ATTR_ROW_BIND_TYPE,
               SQL_BIND_BY_COLUMN, 0);
// Размер набора строк задаем атрибутом оператора
// SQL_ATTR_ROW_ARRAY_SIZE
SQLSetStmtAttr(hstmt, SQL_ATTR_ROW_ARRAY_SIZE,
               ROW_ARRAY_SIZE, 0);
// Устанавливаем атрибут оператора
// SQL_ATTR_ROW_STATUS_PTR для определения массива
// состояний строк
SQLSetStmtAttr(hstmt, SQL_ATTR_ROW_STATUS_PTR,
               RowStatusArray, 0);
// Устанавливаем атрибут оператора
// SQL_ATTR_ROWS_FETCHED_PTR для указания на
// cRowsFetched
SQLSetStmtAttr(hstmt, SQL_ATTR_ROWS_FETCHED_PTR,
               &NumRowsFetched, 0);

// Связываем массив со столбцом ID1
SQLBindCol(hstmt, 1, SQL_C_ULONG,
           ID1Array, 0,
           ID1IndArray);

// Выполняем SQL-оператор SELECT для создания
// результирующего набора
SQLExecDirect(hstmt,
              "SELECT ID1 FROM TBL1",
              SQL_NTS);
```

```
// Выполняем блочную выборку из результирующего набора
// В переменной NumRowsFetched возвращается число
// в действительности выбранных строк
while ((rc = SQLFetchScroll(hstmt,SQL_FETCH_NEXT,0))
        != SQL_NO_DATA)
{
    for (i = 0; i < NumRowsFetched; i++) {

// Отображаем только успешно извлеченные строки (если
// код ответа (rc) равен SQL_SUCCESS_WITH_INFO или
// SQL_ERROR, то строку не выводим.

        if ((RowStatusArray[i] == SQL_ROW_SUCCESS) ||
            (RowStatusArray[i] == SQL_ROW_SUCCESS_WITH_INFO))
        {
            if (ID1IndArray[i] == SQL_NULL_DATA)
                cout<<" NULL ";
            else
                cout<< ID1Array[i];
        }
    }
}
// Закрываем курсор
SQLCloseCursor(hstmt);
```

При использовании связывания по строкам определяется структура, содержащая от одного до трех элементов для каждого столбца извлекаемых данных. Первый элемент предназначается для извлекаемых данных, второй – для длины / индикатора буфера, третий – для индикатора буфера при раздельном сохранении значений длины и индикатора. Массив таких структур должен содержать количество элементов равное числу строк в извлекаемом наборе строк.

Следующий пример иллюстрирует применение связывания по строкам для набора строк, состоящего из двух столбцов.

```
#define ROW_ARRAY_SIZE 30

// Определяем структуру AR_INFO и создаем массив
// структур, содержащий 30 элементов
typedef struct {
    SQLINTEGER    F1;      // Для значения 1 столбца
    SQLINTEGER    F1Ind;   // Для длины/индикатора
    SQLCHAR       F2[11];  // Для значения 2 столбца
    SQLINTEGER    F2LenOrInd;
} ORDERINFO;
AR_INFO TBL_Array[ROW_ARRAY_SIZE]; // Массив структур

SQLINTEGER       NumRowsFetched;
SQLUSMALLINT     RowStatusArray[ROW_ARRAY_SIZE], i;
SQLRETURN        rc;
SQLHSTMT         hstmt;

// Определяем размер структуры, используя атрибут оператора
SQL_ATTR_ROW_BIND_TYPE, и одновременно устанавливаем тип
связывания по строкам
SQLSetStmtAttr(hstmt, SQL_ATTR_ROW_BIND_TYPE,
               sizeof(AR_INFO), 0);

// Используя атрибут оператора SQL_ATTR_ROW_ARRAY_SIZE
// устанавливаем количество строк в извлекаемом наборе
// строк
SQLSetStmtAttr(hstmt, SQL_ATTR_ROW_ARRAY_SIZE,
               ROW_ARRAY_SIZE, 0);

// Используя атрибут оператора SQL_ATTR_ROW_STATUS_PTR
// определяем массив состояния строк
SQLSetStmtAttr(hstmt, SQL_ATTR_ROW_STATUS_PTR,
               RowStatusArray, 0);
```



```

// Устанавливаем атрибут оператора
// SQL_ATTR_ROWS_FETCHED_PTR для указания на
// NumRowsFetched
SQLSetStmtAttr(hstmt, SQL_ATTR_ROWS_FETCHED_PTR,
               &NumRowsFetched, 0);

// Связываем поля структуры первого элемента массива
// со столбцами ID1, Sal и Status
SQLBindCol(hstmt, 1, SQL_C_ULONG,
           &TBL_Array[0].F1,
           0,
           &TBL_Array[0].F1Ind);
SQLBindCol(hstmt, 2, SQL_C_CHAR,
           TBL_Array[0].F2,
           sizeof(TBL_Array[0].F2),
           &TBL_Array[0].F2LenOrInd);
// Выполняем SQL-оператор SELECT для формирования
// результирующего набора
SQLExecDirect(hstmt,
              "SELECT F1, F2 FROM TBL1", SQL_NTS);

// Используя блочный курсор извлекаем установленное число строк
while ((rc = SQLFetchScroll(hstmt, SQL_FETCH_NEXT, 0))
      != SQL_NO_DATA)
{
    // Переменная NumRowsFetched содержит число
    // в действительности извлеченных строк
    for (i = 0; i < NumRowsFetched; i++)
    {
        if (RowStatusArray[i] == SQL_ROW_SUCCESS ||
            RowStatusArray[i] ==
            SQL_ROW_SUCCESS_WITH_INFO) {
            if (TBL_Array[i].F1Ind == SQL_NULL_DATA)
                std::cout<<" NULL      ";
        }
    }
}

```

```
        else      std::cout<< TBL_Array[i].F1;
        if (TBL_Array[i].F2LenOrInd == SQL_NULL_DATA)
            std::cout<< " NULL      ";
        else      std::cout<< TBL_Array[i].F2;
    }
}
// Закрываем курсора
SQLCloseCursor(hstmt);
```

## Тема 5. Механизмы удаленного доступа. Интерфейс ODBC.

### **Объектная модель OLE DB**

OLE DB представляет собой набор COM-интерфейсов (Component Object Model), которые предоставляют приложению-клиенту унифицированный доступ к различным источникам данных.

Можно сказать, что OLE DB это метод доступа к любым данным через стандартные COM-интерфейсы, вне зависимости от типа данных и места их расположения. В качестве данных могут выступать базы данных, простые документы, таблицы Excel и любые другие источники данных. В отличие от доступа, предоставляемого посредством драйверов ODBC, OLE DB позволяет реализовывать доступ к источникам данных, как с применением языка SQL (к SQL-серверам), так и к любым другим произвольным источникам данных.

Средства, предоставляющие доступ к источнику данных с использованием технологии OLE DB, называются *OLE DB провайдерами*. Программы-клиенты, использующие для доступа OLE DB провайдеры, называются *потребителями данных*.

В том случае, если существует только ODBC драйвер для доступа к конкретному источнику данных, то для применения технологии OLE DB можно использовать OLE DB провайдер, предназначенный для доступа к ODBC источнику данных.

Так как архитектура OLE DB основана на COM, то механизм создания результирующих наборов состоит из последовательностей шагов типа: 1. создание объекта -> 2. запрос указателя на интерфейс созданного объекта -> 3. вызов метода интерфейса.

Аналогично действиям, выполняемым после создания результирующего набора при применении технологии ODBC – выполнению связывания, в технологии OLE DB используется механизм аксессоров. *Аксессоры* описывают каким образом данные записываются в область памяти потребителя данных, устанавливая адресное соответствие между областью памяти в буфере потребителя данных и столбцами данных в результирующем наборе. Иногда такой набор связей называют картой столбцов (column map).

### **Объектная модель OLE DB**

Спецификация OLE DB описывает набор интерфейсов, реализуемых объектами OLE DB. Каждый объектный тип определен как набор интерфейсов. Спецификация OLE DB определяет набор интерфейсов базового уровня, которые должны реализовываться любыми OLE DB провайдерами.

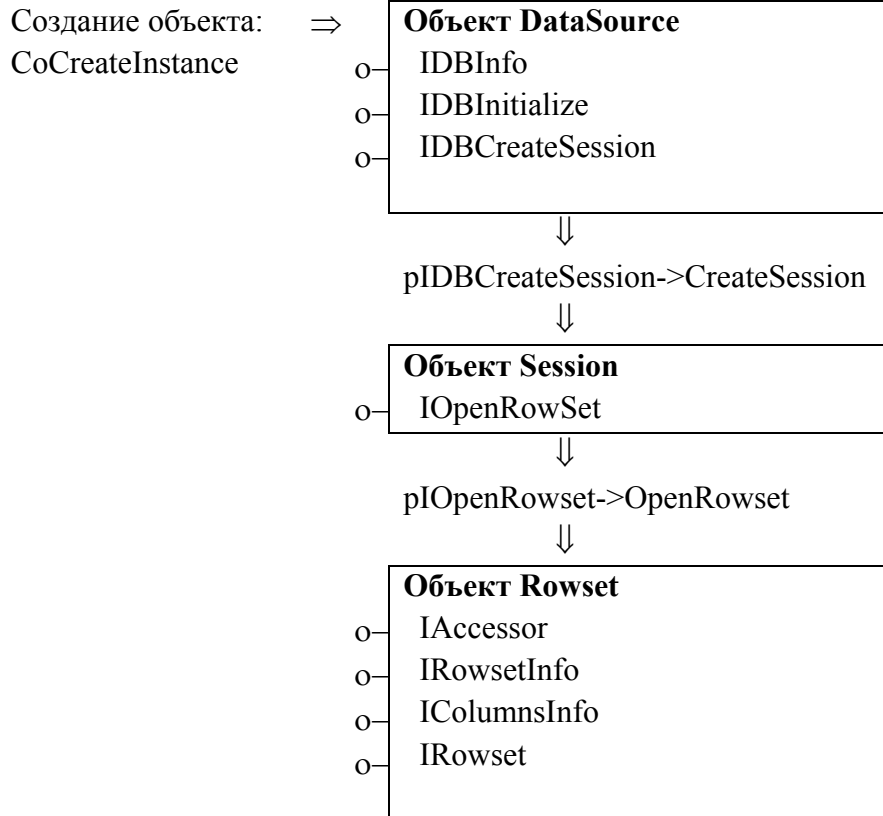
В базовую модель OLE DB входят следующие объекты:

- объект DataSource (источник данных), используемый для соединения с источником данных и создания одного или нескольких сеансов. Этот объект управляет соединением, использует информацию о полномочиях и аутентификации пользователя.
- объект Session (сеанс) управляет взаимодействием с источником данных – выполняет запросы и создает результирующие наборы.

Сеанс также может возвращать метаданные. В сеансе может создаваться одна или несколько команд.

- объект Rowset (результатирующий набор) представляет собой данные, извлекаемые в результате выполнения команды или создаваемые в сеансе.

На следующей схеме приведен пример использования интерфейсов базового уровня для создания результирующего набора.



Спецификация OLE DB определяет объект Command, который предназначается для выполнения текстовой команды. В качестве такой текстовой команды может выступать:

- SQL-оператор
- тест, понимаемый OLE DB провайдером.

При выполнении команды может создаваться результирующий набор. Некоторые OLE DB провайдеры поддерживают работу со схемой (Schema), которая предоставляет метаданные по базе данных. Метаданные становятся доступны как обычные результирующие наборы. В заголовочном файле oledb.h содержатся уникальные идентификаторы всех доступных типов результирующих наборов схемы данных.

Для обеспечения расширенных возможностей управления транзакциями объектная модель OLE DB включает объект Transaction.

OLE DB провайдеры, как и все COM компоненты, регистрируются в реестре Windows. Для поиска информации о зарегистрированных источниках данных используются специальные объекты, называемые нумераторами. Нумератор это обычный COM сервер, позволяющий получить информацию об источниках данных в виде результирующего набора. Для создания такого результирующего набора в объектном типе DataSource специфицирован интерфейс IDBEnumerateSources.

Для каждого объектного типа спецификация OLE DB определяет набор интерфейсов, который должен обязательно быть реализован для данного объекта. Такие интерфейсы отмечаются как [mandatory]. Интерфейсы, которые могут отсутствовать отмечаются как [optional].

Для объекта источник данных специфицирован следующий набор интерфейсов:

```
CoType TDataSource {  
    [mandatory]    interface IDBCreateSession;  
    [mandatory]    interface IDBInitialize;
```

```
[mandatory]    interface IDBProperties;
[mandatory]    interface IPersist;
[optional]     interface IConnectionPointContainer;
[optional]     interface IDBAsynchStatus;
[optional]     interface IDBDataSourceAdmin;
[optional]     interface IDBInfo;
[optional]     interface IPersistFile;
[optional]     interface ISupportErrorInfo;
}
```

Для объекта сеанс специфицирован следующий набор интерфейсов:

```
CoType TSession {
    [mandatory]    interface IGetDataSource;
    [mandatory]    interface IOpenRowset;    // Создание
   // набора данных
    [mandatory]    interface ISessionProperties;
    [optional]     interface IAlterIndex;
    [optional]     interface IAlterTable;
    [optional]     interface IBindResource;
    [optional]     interface ICreateRow;
    [optional]     interface IDBCreateCommand;
    [optional]     interface IDBSchemaRowset;
    [optional]     interface IIndexDefinition;
    [optional]     interface ISupportErrorInfo;
    [optional]     interface ITableCreation;
    [optional]     interface ITableDefinition;    // Для
   // создания таблицы

    [optional]     interface
ITableDefinitionWithConstraints;
```

```

[optional]    interface ITransaction;
[optional]    interface ITransactionJoin;
[optional]    interface ITransactionLocal;
[optional]    interface ITransactionObject;
}

```

Для объекта результирующий набор специфицирован следующий набор интерфейсов:

```

CoType TRowset {
    [mandatory]    interface IAccessor;
    [mandatory]    interface IColumnsInfo;
    [mandatory]    interface IConvertType;
    [mandatory]    interface IRowset; // Последовательное
                                // чтение таблицы
    [mandatory]    interface IRowsetInfo;
    [optional]     interface IChapteredRowset;
    [optional]     interface IColumnsInfo2;
    [optional]     interface IColumnsRowset;
    [optional]     interface IConnectionPointContainer;
    [optional]     interface IDBAsynchStatus;
    [optional]     interface IGetRow;
    [optional]     interface IRowsetChange; // Для
                                // удаления, изменения и добавления
                                // строк в набор данных
    [optional]     interface IRowsetChapterMember;
    [optional]     interface IRowsetCurrentIndex;
    [optional]     interface IRowsetFind;
    [optional]     interface IRowsetIdentity;
    [optional]     interface IRowsetIndex;
}

```

```
[optional]      interface IRowsetLocate;  // Прямое
                // позиционирование на запись набора данных
[optional]      interface IRowsetRefresh; // Для
                // обновления данных в созданном наборе данных
[optional]      interface IRowsetScroll; // Поддержка
                // скроллинга по набору данных
[optional]      interface IRowsetUpdate;
[optional]      interface IRowsetView;
[optional]      interface ISupportErrorInfo;
[optional]      interface IRowsetBookmark;
}
```

Все объекты объектного типа Rowset должны реализовывать следующие интерфейсы:

- интерфейс IRowset, используемый для извлечения строк;
- интерфейс IAccessor, используемый для определения связывания;
- интерфейс IColumnsInfo, предоставляющий информацию о столбцах результирующего набора;
- интерфейс IRowsetInfo, предоставляющий информацию о самом результирующем наборе;
- интерфейс IConvertType, предоставляющий информацию о преобразовании типов данных, поддерживаемых в результирующем наборе.

При реализации доступа к БД посредством OLE DB провайдера сначала следует создать объект данных и установить соединение с базой данных. Далее необходимо создать объект сеанс. И только потом можно создавать результирующий набор.

Механизм создания объекта сеанс приведен на следующей схеме.

|                                                      |
|------------------------------------------------------|
| CoCreateInstanse → DSO    IDBCreateSession → Session |
|------------------------------------------------------|



Результирующий набор может быть создан одним из следующих способов:

- Для объекта сеанс вызывается метод `IOpenRowset::OpenRowset`, выполняющий непосредственное создание результирующего набора (интерфейс `IOpenRowset` должен поддерживаться любым провайдером);
- Для объекта сеанс вызывается метод `IDBCreateCommand::CreateCommand`, создающий объект `Command`. Далее для объекта команда вызывается метод `ICommand::Execute`. (при использовании интерфейса `IMultipleResults` можно работать с несколькими результирующими наборами);
- Вызывается один из следующих методов `IColumnsRowset::GetColumnsRowset`, `IDBSchemaRowset::GetRowset`, `IViewRowset::OpenViewRowset` или `ISourcesRowset::GetSourcesRowset`.

Чтобы результирующий набор, хранимый на сервере, можно было использовать, необходимо выполнить связывание и извлечение данных. Для этого следует определить структуры типа `DBBINDING`, описывающие столбцы и создать аксессор. Далее для получения строк результирующего набора можно использовать различные методы, включая следующие:

- `IRowset::GetNextRows`;
- `IRowsetLocate::GetRowsByBookMarks`;
- `IRowsetLocate::GetRowAt`;
- `IRowsetScroll::GetRowAtRatio`.

В завершение для записи данных в структуру, определенную аксессором вызывается метод `IRowset::GetData`.

Для освобождения строк после их обработки следует вызвать метод `IRowset::ReleaseRows`.

После просмотра всего результирующего набора следует также освободить аксессор, вызвав метод `IRowset::ReleaseAccessor`, и освободить сам результирующий набор, вызвав метод `IRowset::Release`.

Интерфейс `IAccessor` определяет следующие методы:

- AddRefAccessor – увеличивает число ссылок на данный аксессор;
- CreateAccessor – создает аксессор из набора связываний;
- GetBindings – возвращает связывания, установленные данным аксессором;
- ReleaseAccessor – освобождает аксессор.

Для создания аксессора следует запросить интерфейс IAccessor и выполнить следующий код:

```
HRESULT hr=pIAccessor-> CreateAccessor();
```

Метод CreateAccessor имеет следующее формальное описание:

```
HRESULT CreateAccessor (
    DBACCESSORFLAGS    dwAccessorFlags, // Свойства
                                // аксессора и как он используется
    DBCOUNTITEM         cBindings,      // Число связей
                                // в аксессоре
    const DBBINDING     rgBindings[],   // Описание
                                // столбца или параметра
    DBLENGTH            cbRowSize,      // Число байтов,
                                // используемых для одного набора параметров
    HACCESSOR           *phAccessor,    // Указатель
                                // на созданный аксессор
    DBBINDSTATUS        rgStatus[]);    // Массив значений,
                                // определяющий статус
                                // каждого связывания
```

Каждый столбец формируемого результирующего набора или параметр описывается структурой DBBINDING, которая имеет следующее формальное описание:

```
typedef struct tagDBBINDING {
    DBORDINAL           iOrdinal;       // Порядковый номер
```

```

        // столбца или параметра (начиная с 1)
DBBYTEOFFSET  obValue; // Сдвиг в байтах для
        // значения столбца или параметра в буфере
        // (указатель на буфер задается при
        // создании аксессуара)
DBBYTEOFFSET  obLength;
DBBYTEOFFSET  obStatus;
ITypeInfo     *pTypeInfo;
DBOBJECT      *pObject;
DBBINDEXT     *pBindExt;
DBPART        dwPart;
DBMEMOWNER    dwMemOwner;
DBPARAMIO     eParamIO;
DBLENGTH      cbMaxLen;
DWORD         dwFlags;
DBTYPE        wType;
BYTE          bPrecision;
BYTE          bScale;
} DBBINDING;

```

Поле `wType` определяет тип столбца или параметра, который определяется следующим образом:

```

typedef WORD DBTYPE;
enum DBTYPEENUM {
// Следующие значения точно соответствуют VARENUM
// при автоматизации и не могут быть использованы
// как VARIANT.
    DBTYPE_EMPTY = 0,      // Значение отсутствует,
                          // соответствующего типа С нет
    DBTYPE_NULL = 1,      // Значение равно NULL,

```

```
        // соответствующего типа С нет
DBTYPE_I2 = 2,    // Двухбайтовое целое со знаком,
        // соответствует С типу short
DBTYPE_I4 = 3,    // Четырехбайтовое целое со знаком,
        // соответствует С типу long
DBTYPE_R4 = 4,
DBTYPE_R8 = 5,    // Вещественное двойной точности,
        // соответствует С типу Double

DBTYPE_CY = 6,    // Тип для значения Currency
DBTYPE_DATE = 7,  // Тип для значения даты
    // (дата хранится в виде вещественного числа:
    // целочисленная часть определяет дату,
    // а дробная - время)
DBTYPE_BSTR = 8,  // Указатель на строку BSTR
DBTYPE_IDISPATCH = 9, // Указатель на интерфейс
        // IDispatch
DBTYPE_ERROR = 10, // 32-битовый код ошибки
DBTYPE_BOOL = 11,  // Для логического значения
DBTYPE_VARIANT = 12, // Для значения VARIANT
DBTYPE_IUNKNOWN = 13, // Указатель на интерфейс
        // IUnknown
DBTYPE_DECIMAL = 14,
DBTYPE_UI1 = 17, // Однобайтовое беззнаковое целое,
        // соответствует С типу byte
DBTYPE_ARRAY = 0x2000,
DBTYPE_BYREF = 0x4000,
DBTYPE_I1 = 16,
DBTYPE_UI2 = 18,
```



```
// Typedef struct tagDBTIMESTAMP {  
//     SHORT    year;  
//     USHORT   month;  
//     USHORT   day;  
//     USHORT   hour;  
//     USHORT   minute;  
//     USHORT   second;  
//     ULONG    fraction;  
// } DBTIMESTAMP;  
  
DBTYPE_HCHAPTER = 136  
DBTYPE_PROPVARIANT = 138,  
DBTYPE_VARNUMERIC = 139  
};
```

Перед использованием объекта **Command** следует определить, поддерживается ли данный объект. Для этого с помощью метода `QueryInterface` следует запросить интерфейс `IDBCreateCommand` объекта сеанс.

Объект `Command` должен реализовывать следующие интерфейсы:

- `ICommand`
- `IAccessor`
- `ICommandText`
- `IColumnInfo`
- `ICommandProperties`.

Для создания команды вызывается метод `IDBCreateCommand::CreateCommand` объекта сеанс.

Например:

```
ICommandText pICommandText;
```

```

HRESULT hr= pIDBCreateCommand-> CreateCommand(
    NULL, // Если есть агрегирование, то указатель
           // на управляющий IUnknown
    IID ICommandText, // Запрашиваемый интерфейс
    (IUnknown**)& pICommandText); // Указатель на
                                   // запрашиваемый интерфейс

```

Текст, выполняемый командой, устанавливается при вызове метода `ICommandText::SetCommandText`. При этом указывается уникальный идентификатор GUID синтаксиса команды.

Например:

```

pICommandText->SetCommandText(DBGUID_SQL,
    "SELECT f1, f2 FROM TBL1 WHERE f3>10");

```

Для выполнения команды вызывается метод `ICommand::Execute` (этот метод наследуется интерфейсом `ICommandText`).

Например:

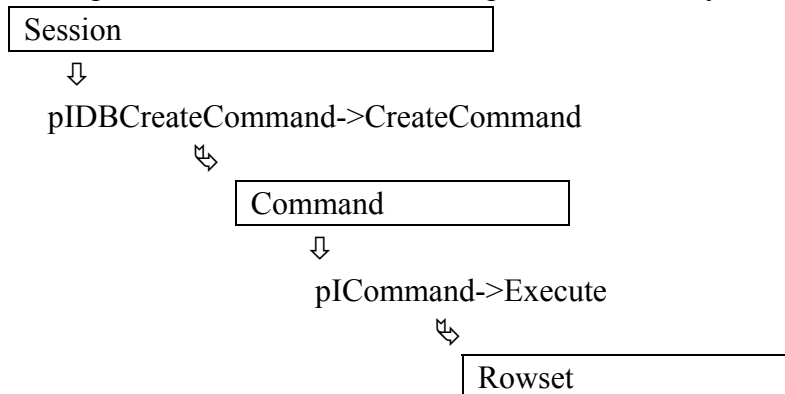
```

ULONG ulRs=0;
IRowset** ppRowsets=NULL;
HRESULT hr= pICommandText->Execute (
    NULL, // если есть агрегирование, то указатель
           // на управляющий IUnknown
    IID IRowset, // Запрашиваемый интерфейс
    NULL, // Указатель на структуру типа
           // struct DBPARAMS {
           //     void *pData;
           //     DB_UPARAMS cParamSets;
           //     HACCESSOR hAccessor;
           // };
    ulRs, // Количество строк, по которым
           // воздействовала команда INSERT, UPDATE

```

```
//или DELETE
(Unknown**)& ppRowsets);    // Указатель на
                             // указатели наборов данных
```

Алгоритм выполнения команды приведен на следующей схеме:



До выполнения команды можно определить поведение создаваемого результирующего набора вызовом метода `ICommandProperties::SetProperties`. Для многократного выполнения запроса и при использовании параметров следует вызвать метод `ICommandPrepare::Prepare`, а затем определить параметры вызовом метода `ICommandWithParameters::SetParameterInfo`. Если в результате выполнения команды возвращается несколько результирующих наборов, то используется метод `IMultipleResults::GetResult`.

Применение OLE DB позволяет поддерживать простые, вложенные и распределенные транзакции.

Объект Session для работы с транзакциями поддерживает следующие интерфейсы:

- интерфейс `ITransactionLocal`. Для начала транзакции вызывается метод `ITransactionLocal::StartTransaction()`. Если этот метод



вызывается из активной транзакции, то открывается новая вложенная транзакция;

- интерфейс `ITransaction`, поддерживающий методы `Abort`, `Commit` и `GetTransactionInfo`;
- интерфейс `ITransactionJoin`, реализующий поддержку распределенных транзакций.

Объект `Transaction` позволяет реализовывать более широкие возможности управления транзакциями, поддерживая следующие интерфейсы:

- `ITransaction`, позволяющий выполнить прерывание транзакции (методы `Abort`, `Commit`, `GetTransactionInfo`);
- `IConnectionPointContainer`, поддерживающий управление точками соединения для соединяемых объектов.

## Тема 6. Применение средств ADO.NET

### ATL

ATL предоставляет OLE DB шаблоны, как C++ шаблоны, для реализации клиентов и серверов OLE DB.

Для реализации клиента OLE DB провайдера можно использовать следующие классы:

- `CDataConnection` – класс, управляющий соединением с источником данных и инкапсулирующий поведение объектов OLE DB источник данных (`DataSource`) и сеанс (`Session`);
- `CDataSource` – класс, соответствующий объекту OLE DB источник данных, предоставляющего соединение с источником данных через OLE DB провайдера. Для одного соединения можно создать несколько объектов сеансов (`CSession`);
- `CEnumerator` – класс, соответствующий объекту OLE DB нумератор, предоставляющему средства для получения наборов данных, содержащих информацию об источниках данных;

- CEnumeratorAccessor – класс, используемый классом CEnumerator для доступа к данным из набора данных нумератора;
- CSession – класс, реализующий сеанс доступа к базе данных.
- CAccessor – класс аксесора, используемый для записей статически связанных с источником данных. Этот класс используется в том случае, если известна структура источника данных;
- CAccessorBase – базовый класс всех классов аксесоров;
- CDynamicAccessor – класс аксесора, используемый для результирующих наборов в том случае, если структура источника данных не известна (создание аксесора может быть определено в режиме выполнения);
- CDynamicParameterAccessor – класс аксесора, используемого в том случае, если типы команды неизвестны. Получить информацию о параметрах можно посредством интерфейса ICommandWithParameters;
- CDynamicStringAccessor – класс аксесора, позволяющий реализовать доступ к источнику данных, если структура базы данных неизвестна;
- CDynamicStringAccessorA – класс аксесора, предоставляющий возможности аналогично классу CDynamicStringAccessor, но с тем ограничением, что данные, доступные из источника данных, были доступны как ANSI-строки данных;
- CDynamicStringAccessorW – класс аксесора, предоставляющий возможности аналогично классу CDynamicStringAccessor, но с тем ограничением, что данные, доступные из источника данных, были доступны как UNICODE -строки данных;
- CManualAccessor – класс аксесора, предоставляющий методы для связывания как столбцов, так и параметров во время выполнения.
- CNoAccessr – используется в том случае, если не нужен класс для связывания параметров или столбцов результирующего набора;
- CXMLAccessor – класс аксесора, предоставляющий возможности аналогично классу CDynamicStringAccessor, но с тем ограничением,

что данные, доступные из источника данных конвертируются в XML-формат;

- CAccessorRowset – инкапсулирует работу с результирующим набором и соответствующим ему аксессором;
- CArrayRowset – позволяет реализовывать доступ к результирующему набору как к массиву;
- CBulkRowset – позволяет выполнять за один вызов функции выборку нескольких строк результирующего набора;
- CNoRowset – может использоваться в том случае, если результирующего набора не возвращается (для CCommand или CTable);
- CRestrictions – используется для задания ограничений для результирующих наборов схемы;
- CRowset – применяется для извлечения данных и управления результирующим набором;
- CStreamRowset – возвращает объект типа ISequentialStream, используемый далее для выборки данных в XML-формате (вызовом метода Read);
- CCommand – применяется для задания и выполнения команд OLE DB; может использоваться для создания результирующих наборов; позволяет задавать параметры;
- CMultipleResults – применяется для команд, создающих несколько результирующих наборов;
- CNoMultipleResults – применяется по умолчанию для команд, создающих только один результирующий набор;
- CTable – используется для доступа к результирующему набору без указания параметров.

Для подключения к базе данных следует создать объект типа CDataSource, затем объект типа CSession.

Классы команды и таблицы позволяют реализовывать доступ к результирующим наборам. Эти классы наследуются от класса `CAccessorRowset`.

В среде Visual Studio.NET мастер ATL OLE DB Consumer позволяет определить какой класс для создания результирующего набора будет использован. Класс `CTable` следует использовать в том случае, если результирующий набор только один и не требуется использование параметров.

Применение класса `CCommand` открывает более широкие возможности:

- создание одного или нескольких результирующих наборов;
- использование параметрических запросов.

Открыть результирующий набор можно вызовом функции `Open`, а при необходимости многократного выполнения команды следует использовать функцию `Prepare`.

Класс `CCommand` имеет три параметра шаблона: тип аксессора, тип результирующего набора и тип результата. Тип результата может указываться значением типа `CNoMultipleResults` (по умолчанию) или `CMultipleResults`.

Технология OLE DB позволяет выполнять редактирование результирующего набора, добавляя, удаляя или изменяя записи.

Для выполнения этих действий в классе `CRowset` реализован интерфейс `IRowsetChange`, предоставляющий следующие методы:

- `SetData` – для изменения записи;
- `Insert` – для добавления записи;
- `Delete` – для удаления записи.

Для редактирования записей должно быть соответствующим образом установлено значение свойства `DBPROP_UPDATABILITY`

результирующего набора. Это можно выполнить как в мастере ATL OLE DB Consumer, так и впоследствии вызовом метода `AddProperty`.

Тип поддерживаемого редактирование указывается следующими константами:

- DBPROPVAL\_UP\_CHANGE – разрешение изменения записи;
- DBPROPVAL\_UP\_INSERT – разрешение вставки новой записи;
- DBPROPVAL\_UP\_DELETE – разрешение удаления записи.

Метод CRowset::SetData устанавливает значение для одного или нескольких столбцов текущей строки.

Метод CRowset::Insert создает новую строку, используя данные из аксессуара и вставляет ее после текущей строки.

## **Объектная модель ADO**

Объектная модель ADO реализована на основе OLE DB. Эта модель в основном предназначена для использования постоянного соединения с базой данных. Извлекаемые наборы данных передаются клиенту посредством маршалинга COM.

Класс Connection реализует физическое соединение с внешним источником данных. Набор данных реализуется объектом Recordset.

Объект Command предназначен для выполнения команд, включая команды, создающие результирующие наборы. Выполняемая команда может создавать более одного результирующего набора. Результирующий набор может быть создан как выполнением оператора SELECT, так и являться результатом выполнения хранимой процедуры.

Объект Command может открыть новое соединение или использовать уже существующее. Это определяется значением его свойства ActiveConnection:

- при задании в качестве значения свойства строки соединения – выполняется открытие нового соединения;
- при указании на существующий объект Connection – используется соединение, установленное посредством данного объекта.

Таким образом, ADO инкапсулирует механизм соединения, применяемый при использовании OLE DB, позволяя выполнить команду без явного создания объектов источник данных и сеанс (не создавая объекта соединения).

Одно и тоже соединение может быть использовано несколькими объектами Command.

Для выполнения запроса посредством объекта Command должен быть вызван метод Execute объекта. Этот метод возвращает объект типа Recordset:

```
cmd.Execute(NumRecords, // Кол-во извлеченных строк
            Parameters,
            CommandOptions); //Тип запроса:
                            // adCmdText – SQL-оператор
                            // adCmdStoreProc – хранимая процедура
```

Например:

```
var Con = Server.CreateObject("ADODB.Connection")
var strCon = "Provider='sqloledb';Data Source=" +
            Request.ServerVariables("SERVER_NAME") + ";" +
            "Initial Catalog='pubs';Integrated Security='SSPI';";
var cmd = Server.CreateObject("ADODB.Command"); // Создание
  // объекта Command
var rs = Server.CreateObject("ADODB.Recordset");
Con.Open(strCon); // Устанавливаем соединение
cmd.CommandText = "MyProc1";
                // CREATE PROCEDURE MyProc1
                // @i1 int
                // AS
                // SELECT f1 from tbl1 WHERE tbl1.f2 = @i1
                // GO

cmd.CommandOptions = adCmdStoredProc;
cmd.CommandTimeout = 15;
```

```
prm = Server.CreateObject("ADODB.Parameter"); // Создание параметра
prm.Type = adInteger; // Тип параметра
prm.Size = 3;
prm.Direction = adParamInput;
prm.Value = 123;
cmd.Parameters.Append(prm);

cmd.ActiveConnection = Con; // Используемое соединение
rs = cmd.Execute(); // Получение результирующего набора

while (!rs.EOF)
{
    rs.MoveNext; // Переход к следующей записи
}
}
if (rs.State == adStateOpen)
    rs.Close;
if (Con.State == adStateOpen)
    Con.Close;
rs = null;
Con = null;
```

Создать результирующий набор можно и непосредственно, используя метод Open объекта Recordset.

Например:

```
var strCon = "Provider='sqloledb';Data Source=" +
    Request.ServerVariables("SERVER_NAME") + ";" +
    "Initial Catalog='pubs';";
var rs = Server.CreateObject("ADODB.Recordset");
```

```
rs.open( "select f1,f2 from tbl1",  
        strCon,                      // Строка соединения  
        adOpenForwardOnly,          // Поведение курсора  
        adLockReadOnly,  
        adCmdText);  
rs.MoveFirst();                      // Переход к первой строке  
while (rs.EOF != true)  
{  
    // ...  
    rs.MoveNext();  
}
```

### **Объектная модель ADO.NET**

Среда Visual Studio .NET предоставляет для доступа к данным объекты ADO.NET. Для доступа и манипулирования данными существуют два пути:

- использовать класс `DataReader` провайдера данных, позволяющий однонаправленный просмотр данных, режим "только чтение";
- использовать класс `DataSet`, реализующий работу с данными, хранимыми в кеше на клиенте.
- 

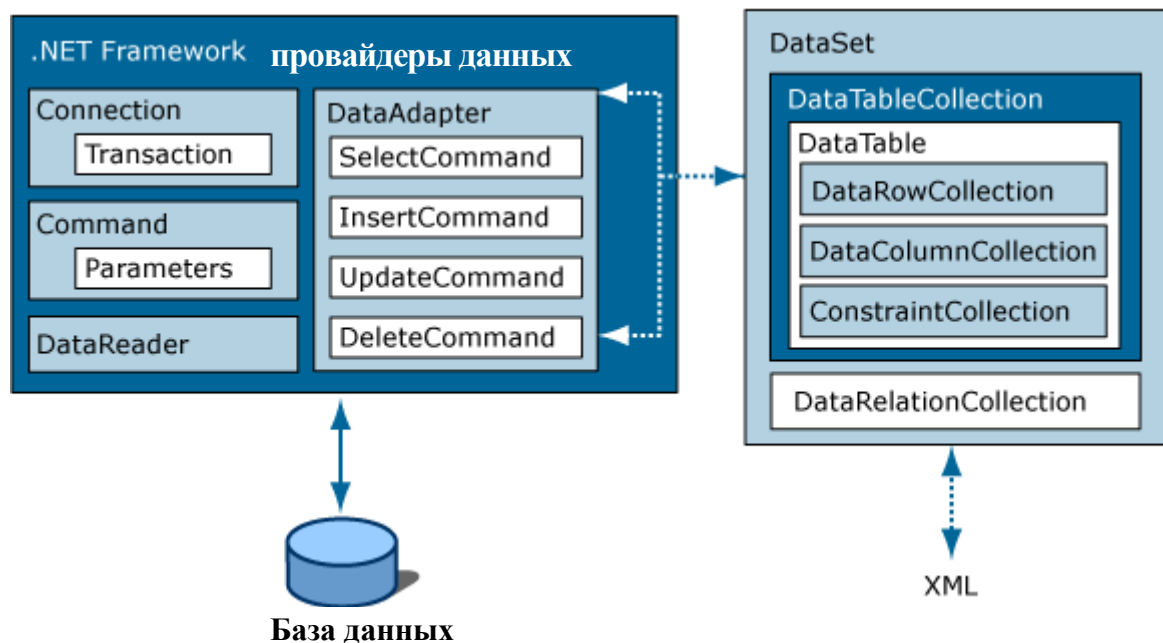
.NET Framework провайдеры данных предоставляют следующие объекты :

- `Connection` - для соединения с базой данных;
- `Command` – для выполнения команд, позволяющих извлекать и изменять данные, выполнять хранимые процедуры;
- `DataReader` – для быстрой выборки данных;
- `DataAdapter` – для реализации моста между объектом `DataSet` и источником данных.



Объект DataSet, в свою очередь, предоставляет собой коллекцию из одного или нескольких объектов DataTable (а DataTable реализуется как коллекция объектов типа DataRowCollection и DataColumnCollection).

На следующей схеме показано взаимодействие между провайдерами данных и объектом набора данных DataSet.



Объект DataReader использует для хранения данных кэш сервера, а объект DataSet – кэш клиента.

.NET Framework позволяет работать с различными провайдерами данных. Причем, для некоторых СУБД, таких как Microsoft SQL Server, применяются провайдеры данных, учитывающие архитектуру конкретной БД.

В следующей схеме приведены объекты .NET Framework провайдеров данных.

| Объекты     | Провайдер данных для OLE DB источника данных | Провайдер данных для SQL Server | Провайдер данных для ODBC источника данных | Провайдер данных для Oracle |
|-------------|----------------------------------------------|---------------------------------|--------------------------------------------|-----------------------------|
| Connection  | OleDbConnection                              | SqlConnection                   | OdbcConnection                             | OracleConnection            |
| Command     | OleDbCommand                                 | SqlCommand                      | OdbcCommand                                | OracleCommand               |
| DataReader  | OleDbDataReader                              | SqlDataReader                   | OdbcDataReader                             | OracleDataReader            |
| DataAdapter | OleDbDataAdapter                             | SqlDataAdapter                  | OdbcDataAdapter                            | OracleDataAdapter           |

### **Соединение с источником данных**

Технология ADO.NET не ориентированна на применение длительных соединений. Вместо этого соединение открывается только на время взаимодействия клиента с источником данных.

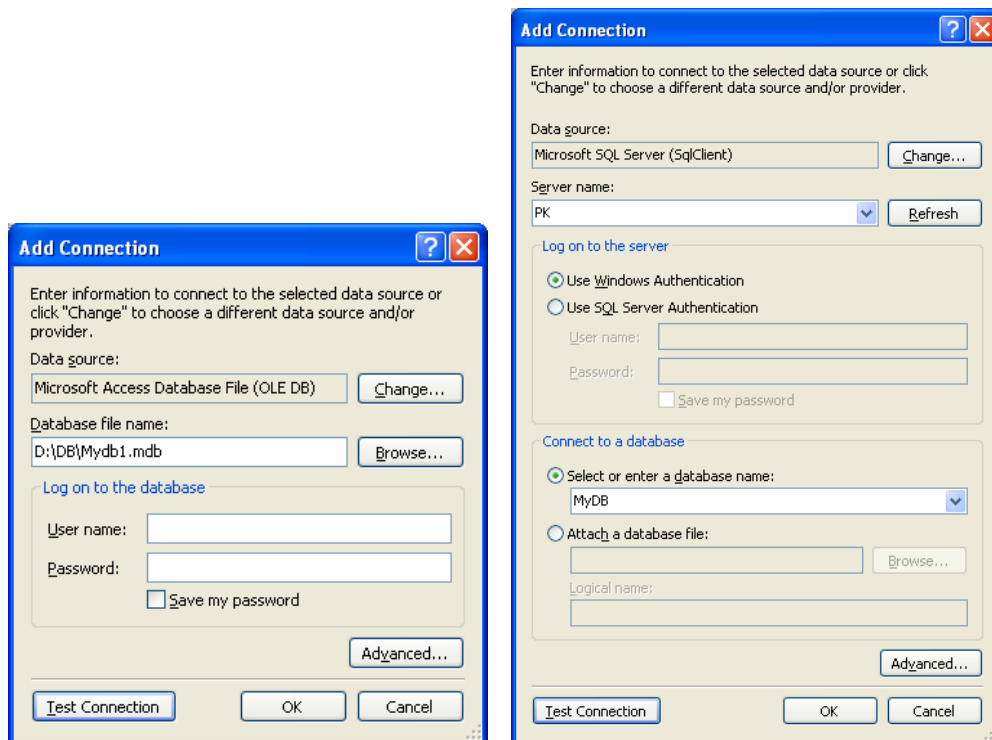
Для того чтобы в среде Visual Studio.NET установить соединение с источником данных следует или выполнить метод Open для объекта соединения, или установить значение свойства Connection объекта Command. В любом случае для этого следует определить строку соединения.

Это можно сделать, используя мастер создания объекта источник данных (команда меню Data | Add New Data Source).

На следующем рисунке представлен диалог выбора источника данных.



В диалоге Add Connection следует ввести необходимую информацию.



Этот диалог позволяет проверить правильность задания всех параметров соединения с источником данных.

Например, при выборе в качестве источника данных базы данных Microsoft Access будет сформирована следующая строка соединения:

Provider=Microsoft.Jet.OLEDB.4.0;Data Source=D:\DB\Mydb1.mdb,

а при выборе источником данных базу данных MyDB cthdthf PK Microsoft SQL Server (базу данных MyDB сервера PK) – строка соединения:

Data Source=PK;Initial Catalog=MyDB;Integrated Security=True.

Создаваемую строку соединения можно добавить к параметрам проекта. В этом случае ее значение можно посмотреть в секции app.config проекта в

окне Solution Explorer. А для указания в качестве параметра при создании объекта соединение требуется использовать следующую нотацию:

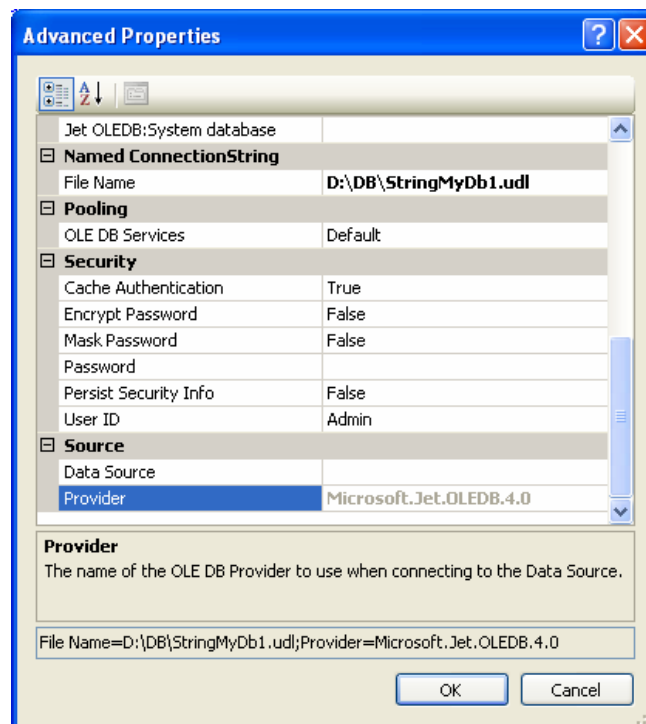
Имя\_проекта.Properties.Settings.Default.строка\_соединения.

Следующий пример содержит код, выполняющий соединение с базой данных:

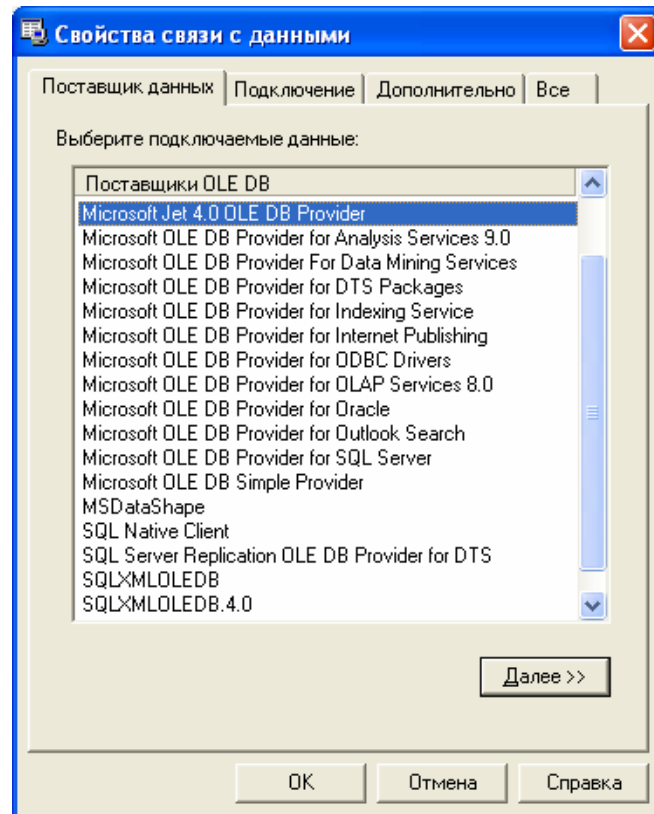
```
using System;
using System.Data;
using System.Data.OleDb;

namespace MyDb1
{
    class Program
    {
        static void Main(string[] args)
        {
            using ( OleDbConnection con = new OleDbConnection (
                // Выбор строки соединения
                MyDb1.Properties.Settings.Default.Mydb1ConnectionString))
            {
                try
                { con.Open(); // Открываем соединение
                }
                catch (Exception ex)
                { Console.WriteLine(ex.Message); }
            }
        }
    }
}
```

При определении строки соединения можно не явно указывать источник данных, а использовать UDL-файл связи с данными. Этот файл указывается свойством FileName:



Используя UDL-файл можно задать поставщика данных и определить подключение к источнику данных.



### **Выполнение SQL-операторов**

Для выполнения SQL-операторов предназначены объекты Command.

Объект OleDbCommand предоставляет следующий набор общедоступных методов:

|                 |                                                                                                                                                 |
|-----------------|-------------------------------------------------------------------------------------------------------------------------------------------------|
| Cancel          | Прерывает выполнение команды.                                                                                                                   |
| Clone           | Создает новый объект OleDbCommand, представляющий собой копию данного.                                                                          |
| CreateObjRef    | Создает объект, содержащий необходимую информацию, требуемую для генерации проку-модуля, используемого для взаимодействия с удаленным объектом. |
| CreateParameter | Создает новый экземпляр объекта OleDbParameter.                                                                                                 |
| Dispose         | Освобождает используемые компонентом ресурсы.                                                                                                   |
| Equals          | Определяет эквивалентность объектов.                                                                                                            |
| ExecuteNonQuery | Выполняет SQL-оператор для соединения (Connection) и возвращает количество строк, на которые воздействовал SQL-оператор.                        |
| ExecuteReader   | Посылает источнику данных текст команды ( свойство CommandText) и генерирует объект OleDbDataReader.                                            |
| ExecuteScalar   | Выполняет запрос и возвращает в виде набора данных первый столбец первой строки.                                                                |
| GetHashCode     | Применяется как хеш-функция для некоторых типов.                                                                                                |

|                           |                                                                                 |
|---------------------------|---------------------------------------------------------------------------------|
| GetLifetimeService        | Возвращает объект, используемый для управления временем жизни данного объекта.  |
| GetType                   | Возвращает тип данного объекта.                                                 |
| InitializeLifetimeService | Настраивает объект, используемый для управления временем жизни данного объекта. |
| Prepare                   | Выполняет компиляцию команды.                                                   |
| ReferenceEquals           | Проверяет эквивалентность объектов.                                             |
| ResetCommandTimeout       | Устанавливает значение по умолчанию для свойства CommandTimeout.                |
| ToString                  | Возвращает имя компонента.                                                      |

### **Объект OleDbDataReader**

Для однонаправленного просмотра данных из серверного курсора следует применять объекты OleDbDataReader, SqlDataReader, OdbcDataReader и OracleDataReader.

Для создания объекта OleDbDataReader следует вызвать метод ExecuteReader объекта OleDbCommand. Доступ к серверному курсору доступен только до тех пор, пока не будет вызван метод Close объекта OleDbDataReader. Также, пока не будет вызван метод Close нельзя выполнять других операций и над используемым объектом OleDbConnection.

При этом изменения, сделанные другим процессом или потоком за время чтения данных, могут быть видимы.

Свойство IsClosed будет доступно только после вызова метода Close объекта OleDbDataReader, а свойство RecordsAffected, доступное и во время



существования серверного курсора, следует вызывать также только после его освобождения.

Например:

```
String cS="Provider=Microsoft.Jet.OLEDB.4.0;Data
                               Source=D:\DB\Mydb1.mdb";
String queryS="select f1,f2 from Tbl1";
using (OleDbConnection con = new OleDbConnection(cS))
{
    OleDbCommand cmd = new
        OleDbCommand(queryS, con);
    con.Open();
    // Создаем серверный курсор:
    OleDbDataReader reader = cmd.ExecuteReader();
    while (reader.Read())
    {
        // Запись в поток в поток вывода
        // значения двух первых столбцов
        // набора данных
        Console.Write(reader[0].ToString());
        Console.Write(" ");
    }
    Console.WriteLine(reader[1].ToString());
    reader.Close();
}
```

Объект `OleDbDataReader` также может использоваться и для доступа к набору данных, сформированному выполнением хранимой процедуры. В этом случае для объекта `OleDbCommand` следует определить тип команды – хранимая процедура, и указать имя вызываемой хранимой процедуры.

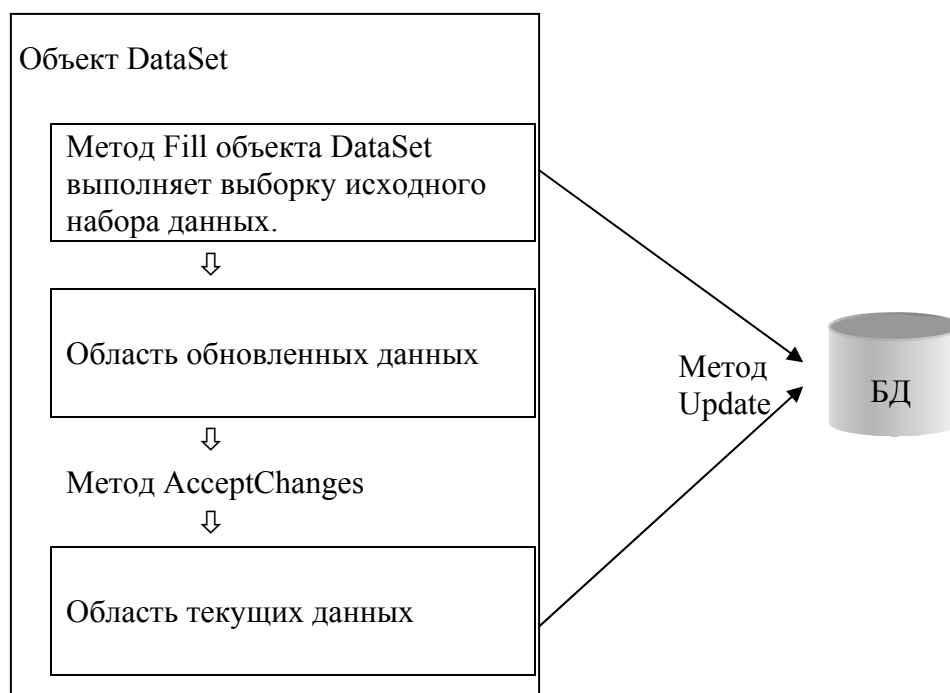
## Объект DataSet

При использовании объектов DataSet создается локальный курсор – кеш данных размещается на клиенте. При этом наряду с самими данными, объект DataSet также содержит и служебную информацию, описывающую эти данные. Набор данных, представимый объектом DataSet, может содержать более одной таблицы.

Набор данных DataSet отличается от набора данных, используемого в ADO. В ADO набор данных это результат выполнения оператора SELECT, а объект DataSet может содержать несколько табличных объектов, включая отношения между ними.

Объект DataSet позволяет работать с отсоединенными данными.

Процесс обновления данных, представленных объектом DataSet, приведен на следующей схеме.



Набор данных может быть сформирован различными способами:

- выборкой из базы данных;
- чтением из XML-файла;
- программное формирование нового набора данных.

Для взаимодействия между набором данных DataSet и базой данных используется класс *адаптера данных*.

Для того чтобы извлечь данные из базы данных и отобразить их в окне формы данных, с использованием объекта DataSet следует:

1. установить соединение с базой данных (Data|Add New Data Source);
2. разместить на форме объект типа DataGridView (страница Data палитры компонентов);
3. перетащить на объект DataGridView набор данных (например, Tbl1) со страницы Data Sources. При этом в код модуля автоматически будет добавлена следующая строка кода:

```
this.tbl1TableAdapter.Fill(this.mydb1DataSet.Tbl1);
```

Объект DataAdapter применяется для выборки данных в один или несколько объектов DataTable. При необходимости отношения между данными могут быть установлены и для разнотипных источников данных (использующих разные провайдеры данных). Отношения между данными задаются объектом DataRelation.

Например:

```
SqlDataAdapter tbl1Adapter = new SqlDataAdapter(  
    "SELECT * FROM dbo.Tbl1",    // Для соединения  
    tbl1Con);    // используем SqlConnection
```

```
OleDbDataAdapter tbl2Adapter = new OleDbDataAdapter(
    "SELECT * FROM Tbl2", // Для соединения
    tbl2Con); // используем OleDbConnection

DataSet tbl_1_2 = new DataSet();
    // Выборка в набор данных двух таблиц
tbl1Adapter.Fill(tbl_1_2, "Tbl1");
tbl2Adapter.Fill(tbl_1_2, "Tbl2");
    // Определение отношений между таблицами
    // набора данных
DataRelation rel = tbl_1_2.Relations.Add(
    "t_1_2",
    tbl_1_2.Tables["Tbl1"].Columns["id1"],
    tbl_1_2.Tables["Tbl2"].Columns["id1"]);

foreach (DataRow pRow in tbl_1_2.Tables["Tbl1"].Rows)
{
    // Записываем в поток вывода значение
    // поля id1 из Tbl1
    Console.WriteLine(pRow["id1"]);
    foreach (DataRow cRow in pRow.GetChildRows(rel))
    {
        // Записываем в поток вывода значение
        // поля id2 из дочерней таблицы tbl2
        Console.WriteLine("\t" + cRow["id2"]);
    }
}
```

Класс DataTable предоставляет для работы с набором данных ряд методов, включая следующие:

|               |                                                                                       |
|---------------|---------------------------------------------------------------------------------------|
| AcceptChanges | Фиксирует все сделанные в таблице изменения с момента последнего вызова этого метода. |
| BeginInit     | Выполняет инициализацию DataTable (при                                                |

|                  |                                                                                                                                                                                                                                                        |
|------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                  | использовании форм или других компонентов).                                                                                                                                                                                                            |
| BeginLoadData    | Отключает использование служебной информации на время загрузки данных..                                                                                                                                                                                |
| Clear            | Удаляет из DataTable все данные.                                                                                                                                                                                                                       |
| Clone            | Копирует все структуры объекта DataTable, включая все схемы и ограничения целостности.                                                                                                                                                                 |
| Compute          | <p>Вычисляет выражение, указываемое первым параметром, для условия, задаваемого вторым параметром. Например:</p> <pre> DataTable table; table = dataSet.Tables["Tbl1"]; object mySum; mySum = table.Compute(     "Sum(f1) ",     "f2 &gt; 20"); </pre> |
| Copy             | Копирует из DataTable структуру и данные.                                                                                                                                                                                                              |
| CreateDataReader | Возвращает объект DataTableReader, соответствующий данным в объекте DataTable.                                                                                                                                                                         |
| Dispose          | Освобождает ресурсы, используемые MarshalByValueComponent.                                                                                                                                                                                             |
| EndInit          | Завершает инициализацию объекта DataTable (используемого для формы или другим компонентом).                                                                                                                                                            |
| EndLoadData      | Определяет, что процесс загрузки данных                                                                                                                                                                                                                |

|                    |                                                                                                                                                                              |
|--------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                    | завершен и все служебная информация набора данных может быть используется.                                                                                                   |
| Equals             | Определяет эквивалентность двух объектов.                                                                                                                                    |
| GetChanges         | Возвращает копию объекта DataTable, содержащую все изменения, сделанные с момента загрузки или последнего вызова метода AcceptChanges.                                       |
| GetDataTableSchema | Метод возвращает объект XmlSchemaSet, содержащий WSDL, описывающий DataTable.                                                                                                |
| GetErrors          | Возвращает массив объектов DataRow, описывающих ошибки.                                                                                                                      |
| GetType            | Возвращает Type для текущего объекта.                                                                                                                                        |
| ImportRow          | Копирует DataRow в DataTable.                                                                                                                                                |
| Load               | Заполняет объект DataTable, используя источник данных, поддерживающий интерфейсIDataReader (объекты DataReader). Извлекаемые строки добавляются к строкам объекта DataTable. |
| LoadDataRow        | Находит и обновляет указанную строку. Если соответствующая строка не найдена, то добавляется новая строка, содержащая указанные данные.                                      |
| Merge              | Объединяет указанный объект DataTable с данным объектом DataTable.                                                                                                           |

|         |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
|---------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| NewRow  | Создает новую строку DataRow в наборе данных.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| ReadXml | <p>Читает схему XML и загружает XML-данные в DataTable.</p> <p>Например:</p> <pre>// Создаем структуру таблицы DataTable table = new     DataTable("TblXml"); DataColumn column = new DataColumn(     "id1",     typeof(System.Int32)); column.AutoIncrement = true; // Добавляем описание столбца table.Columns.Add(column);  column = new DataColumn(     "f1", // Имя столбца     typeof(System.String)); // Тип table.Columns.Add(column);  // Добавляем три строки DataRow row; for (int i = 0; i &lt;= 2; i++) {     row = table.NewRow();     row["f1"] = "string " + i;     table.Rows.Add(row); } // Присваиваем изменения: table.AcceptChanges();  // Записываем объект DataTable // в XML-файл</pre> |

```

table.WriteXml(
    "C:\\TblData.xml",
    XmlWriteMode.WriteSchema);

// Создаем новый объект DataTable
DataTable newTable = new DataTable();

// Читаем из XML-файла
// в объект DataTable
newTable.ReadXml("C:\\TblData.xml");

```

|                 |                                                                                                                                                           |
|-----------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------|
| ReadXmlSchema   | Читает в DataTable схему XML.                                                                                                                             |
| ReferenceEquals | Определяет эквивалентность объектов.                                                                                                                      |
| RejectChanges   | Откат изменений, сделанных в таблице с момента загрузки, или с момента последнего вызова метода AcceptChanges.                                            |
| Reset           | Устанавливает объект DataTable в первоначальное состояние.                                                                                                |
| Select          | Возвращает массив строк, представляющих объекты DataRow.                                                                                                  |
| ToString        | Возвращает строку, содержащую имя таблицы.<br>Например:<br><pre>foreach(DataTable tbl in dataSet.Tables) {     Console.WriteLine(tbl.ToString()); }</pre> |



|                |                                                                                             |
|----------------|---------------------------------------------------------------------------------------------|
| WriteXml       | Записывает текущее содержимое объекта DataTable как XML-данные (в поток вывода или в файл). |
| WriteXmlSchema | Записывает текущую структуру объекта DataTable как схему XML.                               |

Для записи в поток вывода данных объекта DataTable можно использовать следующий код:

```
DataTable table;  
// ...  
foreach (DataRow row in table.Rows)  
{  
    foreach (DataColumn column in table.Columns)  
    { Console.WriteLine (row[column]); }  
}
```

Объект типа DataRow позволяет выполнять редактирование строк. Доступ к этому объекту реализуется через коллекцию Rows объекта DataTable набора данных.

Следующий пример иллюстрирует использование коллекции Rows для внесения изменений в данные.

```
DataTable table = new DataTable("tbl1"); // Создание  
   // объекта DataTable  
DataColumn col = new DataColumn("f1", // Имя поля  
    Type.GetType("System.Int32")); // Тип поля  
  
// Определяем обработку события RowChanged  
table.RowChanged+=new  
    DataRowChangeEventHandler(Row_Changed);  
  
table.Columns.Add(col); // Добавление столбца
```

```
// в объект типа DataTable

// Добавление ограничения UniqueConstraint
table.Constraints.Add(new UniqueConstraint(col));

// Добавление 7 строк.
DataRow newRow; // Объект строка таблицы

for(int i = 0; i < 7; i++)
{
    // При каждой операции вставки будет
    // происходить событие RowChanged
    newRow = table.NewRow();
    newRow[0] = i;
    table.Rows.Add(newRow); // Добавляем новую строку
}
// Присваиваем сделанные изменения
table.AcceptChanges();

// Вызываем перед каждым изменением строки BeginEdit
// и записываем стандартный поток вывода
// первоначальное и текущее значения

foreach(DataRow row in table.Rows)
{
    row.BeginEdit();
    row[0] = (int) row[0] + 1000;
    Console.WriteLine("Первоначальное значение: " +
        row[0, DataRowVersion.Original]);
    // После изменения строка имеет две версии:
    // Current - текущую, и
    // Original - первоначальную,
    // до присвоения изменений версию:
```

```
        // Proposed - предполагаемая
        Console.WriteLine("Предполагаемое значение: " +
            row[0, DataRowVersion.Proposed] + "\n");
    }
    Console.WriteLine("\n");
    // Присваиваем сделанные изменения
    table.AcceptChanges();
    // После вызова BeginEdit устанавливаем для двух строк
    // одинаковые значения, тем самым нарушая
    // ограничение целостности:
    table.Rows[0].BeginEdit();
    table.Rows[1].BeginEdit();
    table.Rows[0][0] = 100;
    table.Rows[1][0] = 100;
    try
    {
        // Вызов метода EndEdit. инициирует проверку
        // ограничения UniqueConstraint
        table.Rows[0].EndEdit();
        table.Rows[1].EndEdit();
    }
    catch (Exception e)
    {
        Console.WriteLine("Тип ошибки {0} occurred.",
            e.GetType());
    }
}
// Обработка события RowChanged
private void Row_Changed(
    object sender,
    System.Data.DataRowChangeEventArgs e)
{
    DataTable table = (DataTable) sender;
```

```
Console.WriteLine("RowChanged " +  
    e.Action.ToString()  
    + "\table" +  
    e.Row.ItemArray[0]);    // Свойство ItemArray  
                           // представляет массив,  
                           // используемый для доступа  
                           // к значениям строки  
}
```

Метод Delete объекта DataRow используется для удаления строки. Свойство RowState позволяет определить текущее состояние строки.

## Тема 7. Классы библиотеки MFC, используемые для доступа к базам данных

### ***Механизмы доступа к СУБД***

Приложения, работающие с базами данных, могут быть реализованы как в рамках архитектуры документ-отображение, так и как приложения-диалоги, или консольные приложения.

Библиотека MFC позволяет реализовывать доступ к базам данных посредством применения ODBC драйверов.

используя;

При этом для реализации доступа к базе данных можно напрямую использовать функции ODBC API.

Для того, чтобы приложение могло отображать данные из таблиц базы данных, вносить в них изменения и передавать изменения в базу данных, достаточно, чтобы был установлен драйвер ODBC для используемой базы данных.

Средства MFC Application Wizard позволяют практически без всякого программирования разрабатывать приложения, выполняющие подключение к таблице базы данных и отображение содержимого ее полей. При этом для реализации доступа к базе данных используется или механизм ODBC или OLE DB.

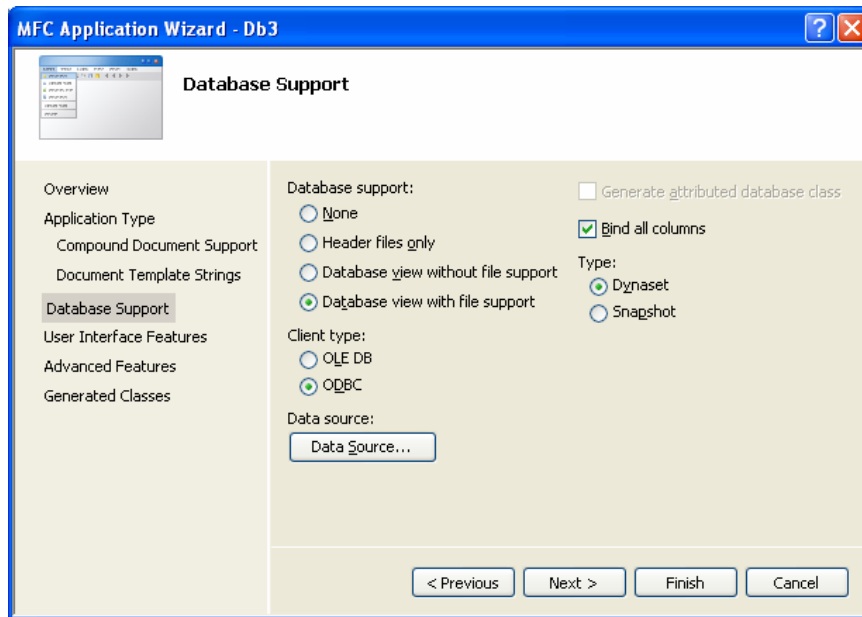
Для того чтобы использовать ODBC-драйверы первоначально следует создать *источник данных* DSN (DataSourceName). Это можно сделать как в момент формирования проекта с помощью MFC Application Wizard, так и используя ODBC32 панели управления Windows.

При создании источника данных определяется имя источника данных, используемое приложениями, выбирается требуемый для подключения к базе данных ODBC-драйвер и указывается местоположение самой базы данных.

Определение имени источника данных выполняется только один раз и затем может быть многократно использовано для создания всех приложений, использующих этот же ODBC-драйвер для этой же базы данных.

Для того чтобы создать приложение, реализующее доступ к базе данных выполните следующие действия:

1. Создайте новый проект. Выберите в качестве шаблона создаваемого проекта MFC Application.
2. Автоматически формируемые приложения, реализующие работу с базой данных, должны поддерживать архитектуру документ-отображение.
3. Перейдите на страницу Database Support и определите поддержку работы с базой данных. Опция Database view without file support обеспечивает формирования кода, выполняющего подключение к источнику данных, но без поддержки механизма сериализации.



4. Выберите механизм доступа (Client type). Опция ODBC обеспечивает включение в проект заголовочного файла AFXDB.H и линкуемых библиотек. При этом никакие классы производные от классов, реализующих работу с базой данных, не создаются. Опция OLE DB обеспечивает включение в проект заголовочных файлов ATLBASE.H, AFXOLEDB.H и ATLPLUS.H.

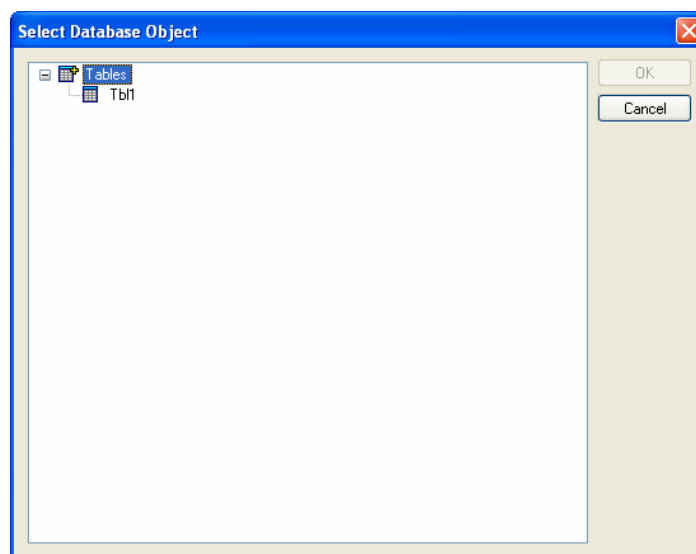
Если для Database Support была установлена опция Database view without file support или Database view with file support, то для механизма ODBC создаваемый класс отображения будет наследоваться от CRecordView и будет ассоциирован с классом результирующего набора, наследуемого от CRecordset. Для механизма OLE DB класс отображения наследуется от COleDBRecordView и ассоциируется с классом, наследуемым от CTable или CCommand.

Тип результирующего набора можно определить одной из следующих опций (требуется поддержка ODBC драйвера):

- `dynaset` – динамический результирующий набор, позволяющий иметь доступ к изменяемой информации результата запроса;
- `snapshot` – результирующий набор (снимок), не отображающий изменений, сделанных после выполнения данного запроса.

Далее следует или выбрать уже существующий источник данных, с которым устанавливается соединение, или создать новый.

После определения используемого источника данных и идентификации пользователя отображается диалог со всеми таблицами и представлениями источника данных, принадлежащих данному пользователю. В этом диалоге можно указать таблицу базы данных, к которой будет выполняться доступ.



На странице `Generated Classes` будет отображен список классов, который MFC Application Wizard автоматически создаст в формируемом приложении.

Класс отображения будет наследован от класса CRecordView, а класс результирующего набора от класса CRecordset.

Далее AppWizard автоматически создаст шаблон приложения. Созданное приложение будет содержать панель инструментов с кнопками для перехода между записями таблицы базы данных, но не будет содержать элементов управления для отображения полей таблицы базы данных.

Отображение полей базы данных

Мастер MFC Application Wizard создает в классе наследуемом от CRecordset переменные члены класса для каждого поля таблицы подключаемой базы данных.

Например:

```
CODBC_1Set::CODBC_1Set(CDatabase* pdb)    // Конструктор класса
    : CRecordset(pdb)    // наследуемого от CRecordset
{
    m_f1 = 0;                // Переменная член класса для 1-го поля
    m_f2 = 0;                // Переменная член класса для 2-го поля
    m_f3 = L"";
    m_f4;
    m_nFields = 4;           // Количество полей
    m_nDefaultType = dynaset;
}
void CODBC_1Set::DoFieldExchange(CFieldExchange* pFX)
{
    pFX->SetFieldType(CFieldExchange::outputColumn);
    RFX_Long(pFX, _T("[f1]"), m_f1);
    RFX_Long(pFX, _T("[f2]"), m_f2);
    RFX_Text(pFX, _T("[f3]"), m_f3);
    RFX_Date(pFX, _T("[f4]"), m_f4);
}
```

Для того чтобы отобразить в форме поля базы данных следует:

1. В редакторе ресурсов для каждого поля таблицы необходимо добавить соответствующий ему элемент управления.

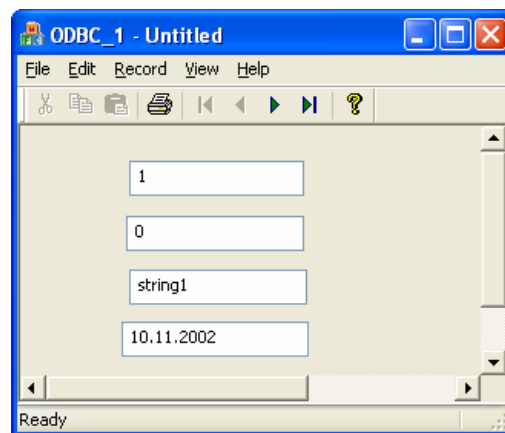


2. В метод DoDataExchange класса отображения следует вставить вызовы методов DDX\_FieldText (или DDX\_FieldCheck, DDX\_FieldRadio, DDX\_FieldSlider и т.п.), выполняющих связь между идентификатором ресурса и переменной членом класса результирующего набора.

Например:

```
void CODBC_1View::DoDataExchange(CDataExchange* pDX)
{
    CRecordView::DoDataExchange(pDX);
    DDX_FieldText(pDX, IDC_EDIT1, m_pSet->m_f1, m_pSet);
    DDX_FieldText(pDX, IDC_EDIT2, m_pSet->m_f2, m_pSet);
}
```

После этого записи базы данных будут отображаться в окне документа в элементах управления окно редактирования.



Библиотека MFC предоставляет удобный интерфейс доступа к базам данных и высокий уровень абстракции от конкретных используемых средств ODBC или DAO. Для каждой открытой базы данных создается отдельный объект *база данных* (класса производного от CDatabase или CDaoDatabase). Используя это объект можно выполнить подключение к базе данных.

Другой объект библиотеки MFC *результатирующий набор* (класса производного от CRecordset или CDaoRecordset), позволяет выполнять запросы и вносить изменения в таблицы базы данных. Классы DAO предоставляют также дополнительные возможности - использование объектов для работы со структурой таблицы (CDaoTableDef) и для сохраняемых запросов (CDaoQueryDef).

Для отображения записей базы данных в рамках архитектуры документ-отображение реализованы классы отображений CRecordView и CDaoRecordView, производные от класса CView.

Рассмотрим пример SDI-приложения, разработанного на основе шаблона приложения созданного MFC Application Wizard . Это приложение реализует доступ к таблице базы данных.

В качестве базового класса объекта отображения будет использован класс CRecordView. В классе документа будет создан набор записей класса производного от класса CRecordset. Обмен данными с таблицей будет реализован посредством RFX-методов в DoFieldExchange.

Рассмотрим наиболее важные фрагменты кода.

*Листинг класса отображения наследуемого от CRecordView:*

```
// ODBC_1View.cpp : реализация класса отображения
#include "stdafx.h"
#include "ODBC_1.h"
#include "ODBC_1Set.h"
#include "ODBC_1Doc.h"
#include "ODBC_1View.h"
IMPLEMENT_DYNCREATE(CODBC_1View, CRecordView)
BEGIN_MESSAGE_MAP(CODBC_1View, CRecordView)
    // Standard printing commands
    ON_COMMAND(ID_FILE_PRINT, CRecordView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_DIRECT, CRecordView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_PREVIEW,
CRecordView::OnFilePrintPreview)
END_MESSAGE_MAP()
CODBC_1View::CODBC_1View()           // Конструктор
    : CRecordView(CODBC_1View::IDD)   { m_pSet = NULL;}
CODBC_1View::~CODBC_1View() { }
```

```
void CODB_1View::DoDataExchange(CDataExchange* pDX)
{
    CRecordView::DoDataExchange(pDX);
    // Определение связи между элементами управления и переменными
    // членами класса результирующего набора
    DDX_FieldText(pDX, IDC_EDIT1, m_pSet->m_f1, m_pSet);
    DDX_FieldText(pDX, IDC_EDIT2, m_pSet->m_f2, m_pSet);
    DDX_FieldText(pDX, IDC_EDIT3, m_pSet->m_f3, m_pSet);
}
BOOL CODB_1View::PreCreateWindow(CREATESTRUCT& cs)
{ // TODO: место для редактирования
  // структуры CREATESTRUCT cs
  return CRecordView::PreCreateWindow(cs);
}
void CODB_1View::OnInitialUpdate()
{
    m_pSet = &GetDocument()->m_ODBC_1Set;    // Связывание документа
  // с результирующим набором
    CRecordView::OnInitialUpdate();
}
BOOL CODB_1View::OnPreparePrinting(CPrintInfo* pInfo)
{ // default preparation
  return DoPreparePrinting(pInfo);
}
void CODB_1View::OnBeginPrinting(CDC* /*pDC*/, CPrintInfo*
/*pInfo*/)
{ // TODO: место для кода, выполняемого перед печатью документа
}
void CODB_1View::OnEndPrinting(CDC* /*pDC*/, CPrintInfo*
/*pInfo*/)
{ // TODO: add cleanup after printing
}
// CODB_1View - поддержка работы с базой данных
CRecordset* CODB_1View::OnGetRecordset()
{ return m_pSet;
}
```

*Листинг класса результирующего набора наследуемого от CRecordset:*

```
// ODBC_1Set.cpp : реализация класса результирующего набора
#include "stdafx.h"
#include "ODBC_1.h"
#include "ODBC_1Set.h"
IMPLEMENT_DYNAMIC(CODBC_1Set, CRecordset)
CODBC_1Set::CODBC_1Set(CDatabase* pdb)          // Конструктор
    : CRecordset(pdb)
{
    m_f1 = 0;          // Инициализация членов класса
    m_f2 = 0;
    m_f3 = L"";
    m_f4;
    m_nFields = 4;
    m_nDefaultType = dynaset;
}
CString CODBC_1Set::GetDefaultConnect()
{
    return
        T("DSN=MySQLDB;DBQ=C:\\C_Project\\db1.mdb;DriverId=25;FIL=MS
        Access;MaxBufferSize=2048;PageTimeout=5;PWD=one;UID=admin;");
}
CString CODBC_1Set::GetDefaultSQL()
{ return _T("[tbl1]");          // Имя просматриваемой таблицы
                                // базы данных
}
void CODBC_1Set::DoFieldExchange(CFieldExchange* pFX)
{
    pFX->SetFieldType(CFieldExchange::outputColumn);
    RFX_Long(pFX, _T("[f1]"), m_f1);
    RFX_Long(pFX, _T("[f2]"), m_f2);
    RFX_Text(pFX, _T("[f3]"), m_f3);
    RFX_Date(pFX, _T("[f4]"), m_f4);
}
```

## Класс CDatabase

Объект CDatabase обеспечивает средства подключения к источнику данных. В качестве источника данных могут выступать Microsoft SQL Server, Microsoft Access, Oracle и т.п. Одновременно в приложении может использоваться несколько активных объектов CDatabase.

Объект CDatabase позволяет устанавливать постоянное соединение с базой данных, которое не предполагается закрывать после каждой операции выборки или обновления данных. Тем самым, если данные предполагается использовать в интерактивном режиме, то время соединения будет достаточно велико. Этот подход противоположен подходу, используемому ADO.NET, где время соединения сведено к минимуму.

Для использования базы данных следует:

1. Создать объект CDatabase.
2. Установить соединение, открыв базу данных (вызвав метод CDatabase::OpenEx или CDatabase::Open).
3. Создать объект CRecordset для операций над подсоединенным источником данных, передав конструктору указатель на CDatabase.
4. Для завершения работы закрыть базу данных, вызвав метод CDatabase::Close. Это также закроет все наборы записей.

Для использования класса CDatabase следует подключить заголовочный файл afxdb.h.

В класс CDatabase включены следующие члены класса:

### CDatabase::m\_hdbc

Указатель на подсоединенный ODBC источник данных.

Например:

```
// Использование m_hdbc для прямого вызова ODBC API.  
// m_db является объектом CDatabase,  
// m_hdbc - член класса CDatabase  
nRetCode = ::SQLGetInfo( m_db.m_hdbc, SQL_ODBC_SQL_CONFORMANCE,  
                        &nValue, sizeof( nValue ), &cbValue );
```

**CDatabase( );**

Конструктор объекта CDatabase.

virtual BOOL **OpenEx** (LPCTSTR *lpszConnectionString*, DWORD *dwOptions* = 0);

**throw( CDBException, CMemoryException );**

Метод открывает базу данных.

При успешном завершении метод возвращает ненулевое значение. Если пользователь щелкает по кнопке Cancel в диалоге, запрашивающем дополнительную информацию, то метод возвращает значение 0. В остальных случаях бросается исключение.

Параметры:

*lpszConnectionString* - определяет ODBC строку подключения. Она включает источник данных и некоторую дополнительную информацию, такую как идентификатор пользователя и пароль.

Например: "DSN=SQLServer\_Source;UID=U1;PWD=user1".

Если указать параметр *lpszConnectionString* равным NULL, то появится стандартный диалог Data Source, позволяющий пользователю выбрать доступный источник данных.

*dwOptions* - по умолчанию параметр равен нулю, что предполагает, что база данных будет открыта с разделяемым доступом и с правами записи; ODBC Cursor Library DLL не будет загружена; ODBC диалог будет появляться только в том случае, если недостаточно информации для выполнения подключения к базе данных. Этот параметр является битовой маской, определяемой комбинацией следующих значений:

CDatabase::openExclusive – в текущей версии MFC-библиотеки не поддерживается: источник данных всегда открывается в разделяемом режиме;

CDatabase::openReadOnly – открытие источника данных только для чтения;

CDatabase::useCursorLib – указывает загрузку библиотеки ODBC Cursor Library DLL, маскирующей некоторую функциональность ODBC-драйвера;

CDatabase::noOdbcDialog – предотвращает появление диалога для подключения источника данных;

CDatabase::forceOdbcDialog – обеспечивает отображение диалога с целью определения информации для ODBC соединения.

Например:

```
// Встраивание объекта CDatabase в производный класс документа
CDatabase m_dbBData( );
// Объект CDatabase: открытие источника данных только на чтение
m_dbBData.OpenEx( _T( "DSN=MYDATASOURCE;UID=U1" ),
                  CDatabase::openReadOnly | CDatabase::noOdbcDialog )
);
```

**virtual void Close();**

Метод закрывает соединение с базой данных. Предварительно следует закрыть все наборы записей ассоциированных с базой данных.

Все операции AddNew и Edit над наборами записей будут прерваны и выполнен откат незавершенных транзакций.

После закрытия соединения объект CDatabase можно использовать для открытия другого источника данных.

**BOOL IsOpen( ) const;**

Метод определяет, установлено ли соединение с базой данных для объекта CDatabase.

**const CString& GetConnect ( ) const;**

Метод возвращает строку подключения, использованную для открытия источника данных.

**CString GetDatabaseName ( ) const;**

Метод возвращает имя текущей подсоединенной базы данных. Это не одно и тоже со значением DSN, указываемым для методов OpenEx и Open, и зависит от ODBC.

В случае возникновения ошибки метод возвращает пустую строку.

**BOOL CanTransact( ) const;**

Метод позволяет определить, можно ли использовать транзакции для базы данных.

**void SetLoginTimeout ( DWORD dwSeconds );**

Метод устанавливает интервал времени, по истечении которого попытка подсоединения к источнику данных будет отменена.

По умолчанию это значение равно 15 секундам.

Отметим, что не все базы данных поддерживают это значение.

**BOOL BeginTrans ( );**

Метод устанавливает начало транзакции.

Транзакция может включать один или несколько вызовов методов AddNew, Edit, Delete, Update для объекта класса CRecordset.

При завершении транзакции вызывается метод CommitTrans для подтверждения всех сделанных изменений в источнике данных, или Rollback -для отмены.

**BOOL CommitTrans ( );**

Метод выполняет завершение транзакции.

Отметим, что для набора записей поддерживается только один уровень транзакций. Вложенные транзакции использовать нельзя.

Например:

```
BOOL CMyDoc::RemoveSt( CString strStID )
{
    // Удаление записей из двух наборов записей
    if ( !m_dbMyBD.BeginTrans( ) ) return FALSE;
    CSet rsSet(&m_dbMyBD); // Создание первого набора записей
    rsSet.m_strFilter = "StID = " + strStID;
    if ( !rsSet.Open(CRecordset::dynaset) ) return FALSE;

    CSet2 rsSet2(&m_dbMyBD); // Создание второго набора записей
    rsSet2.m_strFilter = "StID = " + strStID;
    if ( !rsSet2.Open(CRecordset::dynaset) ) return FALSE;
    TRY // Удаление записей
    {
```



```

        while ( !rsSet.IsEOF( ) )
        {
            rsSet.Delete( );    // Удаление записи из первого
набора
            rsSet.MoveNext( );
            rsSet2.Delete( );    // Удаление записи из второго набора
            m_dbMyBD.CommitTrans( );
        }
        CATCH_ALL(e)
        {
            m_dbMyBD.Rollback( );    // Откат транзакции
            return FALSE;
        }
        END_CATCH_ALL
        rsSet.Close( );    // Закрытие результирующего набора
        rsSet2.Close( );    // Закрытие второго результирующего набора
        return TRUE;
    }
}

```

### **BOOL Rollback ( );**

Метод выполняет откат транзакции.

### **void ExecuteSQL ( LPCSTR *lpSQL* ); throw( CDBException );**

Метод выполняет указанный SQL-оператор.

Выполнение этого оператора не возвращает набора данных.

Например:

```

CString strCmd = "UPDATE tbl1 SET Field1 = 137";
TRY{    m_dbCust.ExecuteSQL( strCmd );
}
CATCH(CDBException, e)
{
    // Код ошибки в e->m_nRetCode
}
END_CATCH

```

## **Класс CRecordset**

Класс CRecordset реализует операции над набором записей, извлеченных из источника данных.

Результирующий набор может использоваться в двух режимах:

- динамический набор записей (dynasets), состояние которого синхронизируется с изменениями, сделанными другими пользователями;
- статический набор записей (snapshots).

Класс CRecordset позволяет прокручивать записи результирующего набора, изменять записи и выполнять блокировку записей, сортировать записи, определять фильтр выбора записей из источника данных.

Для использования набора записей из базы данных следует:

1. Создать объект класса CRecordset, передав конструктору в качестве параметра указатель на объект CDatabase или NULL;
2. Вызвать метод Open и определить режим использования набора данных: динамический или статический.

Метод Open открывает результирующий набор. После этого набор записей можно просматривать и редактировать.

Для разрушения объекта результирующий набор следует вызвать метод Close.

В классе CRecordset реализованы следующие члены класса:

#### **CRecordset::m\_hstmt**

Содержит указатель на структуру данных типа HSTMT для SQL-оператора. При доступе через ODBC каждый SQL-оператор ассоциируется с переменной, указывающей на структуру типа.

Эта переменная используется непосредственно в методе CDatabase::ExecuteSQL.

Значение переменной m\_hstmt будет доступно только после выполнения метода Open.

#### **CRecordset::m\_nFields**

Содержит количество полей в результирующем наборе.

Это число должно соответствовать количеству полей, зарегистрированных методом DoFieldExchange или DoBulkFieldExchange после вызова SetFieldType с параметром CFieldExchange::outputColumn.

При динамическом встраивании столбцов значение `m_nFields` следует увеличивать в соответствии с количеством вызванных в методе `DoFieldExchange` RFX функций.

#### **CRecordset::m\_nParams**

Переменная содержит количество параметров, используемых в запросе.

Параметры указываются символом `?` в строке `m_strFilter` или строке `m_strSort`.

При добавлении переменной члена класса для параметра запроса в конструкторе следует увеличить значение переменной `m_nParams`.

#### **CRecordset::m\_pDatabase**

Переменная содержит указатель на объект `CDatabase` посредством которого для данного результирующего набора установлено соединение с базой данных.

#### **CRecordset::m\_strFilter**

Переменная содержит условие, указываемое во фразе `WHERE` SQL-оператора.

Значение этой переменной следует присвоить сразу после создания объекта результирующий набор и до выполнения метода `Open`. Строка условия может включать параметры, указываемые символом `?`.

Например:

```
CMySet rsMySet( NULL ); // Класс CMySet наследуется от
CRecordset
// Определение фильтра (условия во фразе WHERE)
rsMySet.m_strFilter = "field2 = 'aaa'";
// Выполнение запроса - открытие результирующего набора
rsMySet.Open( CRecordset::snapshot, "MyTbl1" );
```

#### **CRecordset::m\_strSort**

Переменная содержит значение, указываемое во фразе `ORDER BY` SQL-оператора.

**CRecordset ( CDatabase\* pDatabase = NULL);**

Конструктор объекта `CDatabase`.

Если параметр *pDatabase* указывает базу данных, но для нее не был вызван метод *Open*, то метод *Open* объекта *CRecordset* попытается установить соединение с источником данных. Если указан параметр *NULL*, то объект *CDatabase* будет создан и подключен автоматически к источнику данных (на основе информации об источнике данных, указанной или при использовании MFC Application Wizard, или при создании производного класса в ClassWizard).

```
virtual BOOL Open ( UINT nOpenType =  
AFX_DB_USE_DEFAULT_TYPE, LPCTSTR lpszSQL = NULL, DWORD  
dwOptions = none );  
throw( CDBException, CMemoryException );
```

Метод открывает результирующий набор.

*nOpenType* - значение по умолчанию равно

*AFX\_DB\_USE\_DEFAULT\_TYPE*. Этот параметр может быть задан одним из следующих значений:

*CRecordset::dynaset* - результирующий набор с двунаправленным просмотром. Поля, порядок записей и значения определяются при открытии набора записей. Изменения значений сделанные другими пользователями отображаются при следующей операции выборки.

*CRecordset::snapshot* - результирующий набор с двунаправленным просмотром. Поля, порядок записей и значения определяются при открытии набора записей. Изменения сделанные другими пользователями не доступны, пока набор записей не будет закрыт и опять повторно открыт.

*CRecordset::dynamic* - результирующий набор с двунаправленным просмотром. Изменения сделанные другими пользователями для полей, порядка и значений данных видны после следующей операции выборки. отметим, что не все ODBC поддерживают такой тип набора записей.

*CRecordset::forwardOnly* результирующий набор с доступом только на чтение и просмотром только вперед.

По умолчанию используется значение *CRecordset::snapshot*.

*lpzSQL* - указатель строки, которая может содержать или NULL, или имя таблицы, или SQL-оператор SELECT (возможно с фразами SQL WHERE и ORDER BY).

Отметим, что порядок столбцов в результирующем наборе должен соответствовать порядку RFX методов, вызываемых в методе DoFieldExchange.

*dwOptions* - битовая маска, определяемая допустимой комбинацией следующих значений:

- CRecordset::none
- CRecordset::appendOnly
- CRecordset::readOnly
- CRecordset::optimizeBulkAdd
- CRecordset::useMultiRowFetch
- CRecordset::skipDeletedRecords
- CRecordset::useBookmarks
- CRecordset::noDirtyFieldCheck
- CRecordset::executeDirect
- CRecordset::useExtendedFetch
- CRecordset::userAllocMultiRowBuffers.

Отметим, что для объекта CRecordset всегда используется разделяемый доступ и в отличие от объекта CDaoRecordset нельзя использовать эксклюзивный доступ.

**BOOL CanAppend ( ) const;**

Метод определяет, разрешено ли в данный открытый набор записей добавлять новые записи.

**const CString& GetTableName ( ) const;**

Метод возвращает имя таблицы, для которой был выполнен запрос (получен результирующий набор) или пустую строку.

**const CString& GetSQL ( ) const;**

Метод возвращает SQL-оператор, использованный для получения открытого в данный момент набора записей.

**BOOL IsEOF ( ) const;**

Метод позволяет определить, достигнут ли конец набора записей, а также является ли набор записей пустым.

Сразу после открытия набора записей текущей устанавливается первая запись. Если набор записей пуст, то вызов IsEOF вернет значение 0.

**BOOL IsBOF ( ) const;**

Метод позволяет определить, достигнуто ли начало набора записей.

Например:

```
rsSet.Open( ); // Набор записей открыт: текущая запись - первая
if( rsSet.IsBOF( ) ) return; // Набор записей пуст
while ( !rsSet.IsEOF( ) ) // Просмотр всех записей от
                        // начала до конца

    rsSet.MoveNext( );
rsSet.MoveLast( ); // Переход к последней записи
while( !rsSet.IsBOF( ) ) // Просмотр всех записей от
                        // конца до начала

    rsSet.MovePrev( );
rsSet.MoveFirst( ); // Переход к первой записи
```

**virtual void AddNew ( );throw( CDBException );**

Метод выполняет подготовку для добавления новой записи (этот метод (также как и Delete, Edit, Update) нельзя использовать при выполнении выборки набора записей). Для того чтобы увидеть добавленную новую запись для статического набора записей, следует вызвать метод Requery.

Для динамического набора записей новые записи добавляются в конец.

Этот метод подготавливает новую пустую запись, используя поля набора записей. Затем можно установить значения для полей новой записи (метод Edit нельзя использовать в данном случае - он может быть вызван только для существующих записей). Для внесения изменений в сам источник данных следует выполнить метод Update.

Отметим, что если прокрутить новую запись до вызова Update, то запись будет потеряна.

Если используемый источник данных поддерживает транзакции, то вызов метода AddNew может быть частью транзакции.

**void CancelUpdate ();**

Метод отменяет сделанные операциями Edit или AddNew изменения с момента последнего вызова метода Update.

**virtual void Delete (); throw( CDBException );**

Метод удаляет текущую запись и устанавливает в Null значения полей набора записей. После вызова этого метода следует выполнить вызов Move для перехода от удаленной записи (потом к ней уже нельзя вернуться).

Метод Delete может быть частью транзакции.

**virtual void Edit (); throw( CDBException, CMemoryException );**

После вызова этого метода можно редактировать текущую запись. Операция редактирования завершается вызовом метода Update, сохраняющем сделанные изменения в источнике данных.

Если вызвать метод Edit вторично до вызова метода Update, то будут восстановлены значения полей текущей записи, которые были до начала редактирования.

Например:

```
rsSet.Edit( );           // Начало операции изменения записи:
    // теперь можно редактировать поля набора записей -
    // члены класса
rsSet.m_dwCITY = "Moscow";
rsSet.m_strName = "Olga";
if( !rsCustSet.Update( ) )    // Завершение операции изменения
                               // записи
```

**virtual BOOL Update ();throw( CDBException );**

Метод возвращает ненулевое значение, если запись источника данных была успешно обновлена. Если нет измененных полей, то метод возвращает

значение 0. Если нет обновляемых записей, или изменено более одной записи, то бросается исключение.

```
void MoveFirst (); throw( CDBException, CMemoryException );  
void MoveLast (); throw( CDBException, CMemoryException );  
void MoveNext(); throw( CDBException, CMemoryException );  
void MovePrev(); throw( CDBException, CMemoryException );
```

Эти методы позволяют перемещать указатель текущей записи.

```
virtual void DoFieldExchange ( CFieldExchange* pFX );throw(  
CDBException );
```

Этот метод вызывается для автоматического обмена данными между членами класса - полями результирующего набора (текущей записи) и соответствующими полями текущей записи в источнике данных.

Параметры:

*pFX* - Указатель на объект CFieldExchange.

Если реализована выборка набора записей, состоящего из более чем одной записи, то используется метод DoBulkFieldExchange. Для реализации выборки нескольких записей следует для параметра dwOptions в методе Open установить значение CRecordset::useMultiRowFetch.

Отметим, что метод DoFieldExchange доступен только для объектов производного класса от класса CRecordset. Если набор записей создан непосредственно как объект класса CRecordset, то следует использовать метод GetFieldValue.

Обмен данными с источником данных, называемый *RFX-обменом* (record field exchange), работает в двух направлениях: из полей объекта набор записей в поля источника данных и обратно.

Обычно для того, чтобы использовать этот метод достаточно в производном классе для результирующего набора определить имена и тип полей данных - членов класса. ClassWizard самостоятельно вставит код в переопределяемый метод DoFieldExchange.

Например:

```
void CSet::DoFieldExchange(CFieldExchange* pFX)
```



```
{  
    pFX->SetFieldType(CFieldExchange::outputColumn);  
    RFX_Text(pFX, "Name", m_strName);    // Вызов RFX-метода  
    RFX_Int(pFX, "Age", m_wAge);  
}
```

### **Класс CRecordView**

Объект отображение CRecordView предоставляет средства для просмотра полей базы данных в элементах управления.

Отображение создается на основе шаблона документа, используя элементы управления, добавленные в ресурс шаблона диалога.

Объект CRecordView использует DDX-обмен данными и RFX-обмен данными для реализации обмена между тремя наборами данных: элементами управления, полями результирующего набора и записями источника данных.

Дополнительно класс CRecordView поддерживает по умолчанию реализацию просмотра набора записей и перехода к первой, последней, следующей и предыдущей записям в текущем отображении.

При использовании для создания шаблона приложения MFC Application Wizard автоматически будут созданы ресурс меню и панели инструментов, содержащей кнопки прокрутки набора записей. При программировании вручную, используя ClassWizard, эти ресурсы надо создавать самостоятельно в редакторе меню и в редакторе битовых изображений.

Класс CRecordView предоставляет следующие члены класса:

**CRecordView( LPCSTR *lpszTemplateName* );**

**CRecordView( UINT *nIDTemplate* );**

Конструктор объекта CRecordView.

Параметры:

*lpszTemplateName* - строка, содержащая имя ресурса шаблона диалога.

*nIDTemplate* - ID ресурса шаблона диалога, используемого отображением.

Метод `CRecordView::OnInitialUpdate` вызывает метод `UpdateData`, который в свою очередь вызывает метод `DoDataExchange`.

Этот первоначальный вызов `DoDataExchange` соединяет элементы управления `CRecordView` с полями данных - членами класса `CRecordset`.

Отметим, что эти члены класса не могут быть использованы до вызова метода базового класса `CFormView::OnInitialUpdate`.

При использовании `ClassWizard` он определит значение типа `enum CRecordView::IDD` и укажет его в списке инициализации для конструктора.

Например:

```
CMyRecordView::CMyRecordView() : CRecordView( IDD_MY_RES_DLG)
{
    // Инициализация переменных членов класса
}
```

**virtual CRecordset\* OnGetRecordset ( ) = 0;**

Метод возвращает указатель на объект результирующий набор, ассоциированный с отображением, при его успешном создании, и `NULL` в противном случае.

Этот метод следует переопределить для получения набора записей и возвращения указателя на него.

`ClassWizard` самостоятельно вставляет переопределение этого метода.

**BOOL IsOnFirstRecord ( );**

**BOOL IsOnLastRecord(**

Эти методы позволяют определить, является ли текущая запись первой или последней.

**virtual BOOL OnMove ( UINT *nIDMoveCommand* ); throw( CDBException );**

Этот метод используется для перехода к другой записи результирующего набора.

Параметры:

*nIDMoveCommand* - определяет ID одной из следующих стандартных команд:

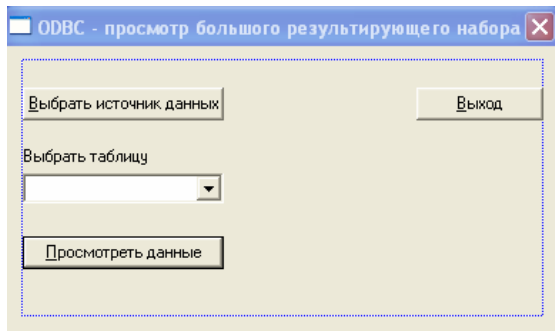
ID\_RECORD\_FIRST  
ID\_RECORD\_LAST  
ID\_RECORD\_NEXT  
ID\_RECORD\_PREV.

По умолчанию OnMove обновляет текущую запись источника данных, если пользователь внес изменения в элементы управления, используемые для отображения записи.

Приведем пример приложения с архитектурой документ-отображение, использующего класс CRecordset для динамического получения результирующего набора в зависимости от выбираемого источника данных и таблицы SQL-сервера.

В главном файле приложения следует создать объект приложения и переопределить метод InitInstance. Это реализуется следующим образом:

```
CDbApp theApp;  
BOOL CDbApp::InitInstance()  
{  
    CDbDlg dlg;  
    m_pMainWnd = &dlg;  
    int nResponse = (int)dlg.DoModal(); // Отображение модального  
    if (nResponse == IDOK)             // диалога  
    {  
    }  
    else if (nResponse == IDCANCEL)  
    {  
    }  
    return FALSE;  
}
```



Внешний вид ресурса диалога, используемого классом CDbDlg

Класс диалога CDbDlg реализует обработку сообщения для трех командных кнопок и одного списка. Для обработки сообщения BN\_CLICKED элемента управления с заголовком "Выбрать источник данных" вызывается метод OnGetdatasource. В этом методе выполняется отображения стандартного диалога для выбора ODBC источника данных. Для обработки сообщения BN\_CLICKED элемента управления с заголовком "Просмотреть данные" вызывается метод OnGetData. В этом методе определяется имя выбранной таблицы базы данных, формируется SQL-оператор, создается результирующий набор и открывается диалог, отображающий данные. Это реализуется следующим образом:

```
void CDbDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    DDX_Control(pDX, IDC_TABLE_COMBO, m_cbTable);
    DDX_Control(pDX, IDC_GETDATASOURCE, m_buttonGetDataSource);
    DDX_Control(pDX, ID_QUIT_SELECT, m_buttonQuit);
    DDX_Control(pDX, ID_GETDATA, m_buttonGetData);
}
BEGIN_MESSAGE_MAP(CDbDlg, CDialog)          // Таблица сообщений
    ON_WM_SYSCOMMAND()
    ON_WM_PAINT()
    ON_WM_QUERYDRAGICON()
    ON_BN_CLICKED(IDOK, OnOk)
```

```

    ON_BN_CLICKED(IDC_GETDATASOURCE, OnGetdatasource)
    ON_BN_CLICKED(ID_QUIT_SELECT, OnQuitSelect)
    ON_BN_CLICKED(ID_GETDATA, OnGetData)
END_MESSAGE_MAP()
void CDbDlg::OnOk() { }
void CDbDlg::OnCancel() { CDialog::OnCancel(); }
void CDbDlg::OnGetdatasource()
{
    m_cbTable.ResetContent(); // Очистка списка таблиц
    if (m_db.IsOpen()) // Если соединение открыто, то закрыть его
        m_db.Close();
        // Отображение диалога ODBC connection:
    if (!m_db.OpenEx(NULL, CDatabase::forceOdbcDialog))
    {
        // После того как соединение установлено сделать доступными
        // элементы управления с идентификаторами IDC_TABLE_COMBO и
        // ID_GETDATA
        GetDlgItem(IDC_TABLE_COMBO)->EnableWindow(FALSE);
        GetDlgItem(ID_GETDATA)->EnableWindow(FALSE);
        return;
    }
    CTables rs(&m_db); // Получение списка таблиц
    // Класс CTables это вспомогательный класс, наследуемый от
    CRecordset
    rs.Open(NULL, NULL, NULL, "TABLE");
    CString strTableRef;
    while (!rs.IsEOF()) // Перебор имен всех таблиц
    {
        // Создание строки, содержащей имя таблицы и владельца
таблицы
        strTableRef = _T("");
        if (!rs.m_strTableOwner.IsEmpty())
            strTableRef += rs.m_strTableOwner + _T("].");
        strTableRef += rs.m_strTableName + _T("]");
        m_cbTable.AddString(strTableRef); // Добавление строки в
        // список соответствующий ресурсу IDC_TABLE_COMBO
        rs.MoveNext();
    }
}

```

```

    rs.Close(); // Закрытие результирующего набора со списком
таблиц
    GetDlgItem(IDC_TABLE_COMBO)->EnableWindow(TRUE);
    GetDlgItem(ID_GETDATA)->EnableWindow(TRUE);
}
void CDbDlg::OnQuitSelect() { OnCancel();}
void CDbDlg::OnGetData()
{
    CDynamicSet rs(&m_db); // Результирующий набор
    CDataDialog dlgData; // Диалог для отображения данных
    // Таблица отображается в элементе управления List control
    int nCurSel = m_cbTable.GetCurSel(); // Индекс выбранного
    // элемента списка

    if (nCurSel == CB_ERR)
    {
        CString strError;
        strError.LoadString(IDS_ERROR_NOTABLE);
        AfxMessageBox(strError);
        return;
    }
    CString strSQL;
    m_cbTable.GetLBText(nCurSel, strSQL); // Получение имени
    // просматриваемой таблицы
    strSQL = T("SELECT * FROM ") + strSQL; // SQL-оператор
    rs.Open(CRecordset::snapshot, strSQL, // Открытие
    // результирующего набора
    CRecordset::readOnly | CRecordset::useMultiRowFetch);
    dlgData.SetRecordset(&rs); // Результирующий набор для
    диалога
    dlgData.DoModal(); // Отображение модального диалога
    rs.Close();
}

```

Класс CDataDialog, реализующий диалог с отображением результирующего набора в виде таблицы, имеет следующее описание:

```

class CDataDialog : public CDialog
{
public:
    CDataDialog(CWnd* pParent = NULL);
    enum { IDD = IDD_GETDATA_DAILOG };
}

```

```

CListCtrl m_listData; // Переменная для элемента
                        // управления список
protected:
virtual void DoDataExchange(CDataExchange* pDX);
protected:
    CDynamicSet* m_prs;           // Переменная для набора данных
    afx_msg void OnFirst();       // Обработчики сообщений для кнопок
    afx_msg void OnLast();
    afx_msg void OnNext();
    afx_msg void OnPrev();
    DECLARE_MESSAGE_MAP()
public:
    void SetRecordset(CDynamicSet* prs);
    virtual int DoModal();
    void FillData();
    virtual BOOL OnInitDialog();
};

```

Класс CDataDialog выполняет заполнение списка данными результирующего набора и обрабатывает сообщения от кнопок, используемых для порционного отображения (по 10 записей) данных. Это реализуется следующим образом:

```

void CDataDialog::SetRecordset(CDynamicSet* prs)
{ ASSERT(prs != NULL);
  m_prs = prs;           // Указатель на набор данных
}
int CDataDialog::DoModal()
{ ASSERT(m_prs != NULL);
  return (int)CDialog::DoModal();
}
BOOL CDataDialog::OnInitDialog()
{BOOL bReturn = CDialog::OnInitDialog();
// Добавление столбцов
ASSERT(m_prs->IsOpen());
  CODBCFieldInfo info;
  int nColumns = m_prs->GetODBCFieldCount(); // Определение

```

```

// числа столбцов в таблице
// Для каждого столбца определение имени поля
// и добавления его в заголовок
for (int nNum = 0; nNum < nColumns; nNum++)
{
    m_prs->GetODBCFieldInfo(nNum, info);
    if (m_listData.InsertColumn(nNum, info.m_strName,
                               LVCFMT_LEFT, 80) != nNum)
    {
        ASSERT(FALSE);
        return;
    }
}
FillData(); // Заполнение столбцов данными
return bReturn;
}

void CDataDialog::FillData()
{
    ASSERT(m_prs->IsOpen());
    m_listData.DeleteAllItems(); // Удаление элементов списка
    // Проверка есть ли данные в результирующем наборе
    if (m_prs->IsEOF() && m_prs->IsBOF())
    {
        // Сделать все командные кнопки недоступными
        GetDlgItem(IDC_FIRST)->EnableWindow(FALSE);
        GetDlgItem(IDC_LAST)->EnableWindow(FALSE);
        GetDlgItem(IDC_NEXT)->EnableWindow(FALSE);
        GetDlgItem(IDC_PREV)->EnableWindow(FALSE);
        // Отобразить сообщение об ошибке
        CString strError;
        strError.LoadString(IDS_ERROR_NODATA);
        AfxMessageBox(strError);
        return;
    }
    else
    {
        // Сделать все командные кнопки доступными
        GetDlgItem(IDC_FIRST)->EnableWindow(TRUE);
        GetDlgItem(IDC_LAST)->EnableWindow(TRUE);
        GetDlgItem(IDC_NEXT)->EnableWindow(TRUE);
        GetDlgItem(IDC_PREV)->EnableWindow(TRUE);
    }
}
```



```

    }
    long* rgLength;
    LPSTR rgData;
    CString strData;
    int nFields = m_prs->GetODBCFieldCount();
    int nRowsFetched = m_prs->GetRowsFetched();
    // Отображение десяти строк из набора данных
    for (int nField = 0; nField < nFields; nField++)
    {
        // Определение правильных данных и длины массива
        // m_ppvData - член класса наследуемого от CRecordset
        rgData = (LPSTR)m_prs->m_ppvData[nField];
        rgLength = (long*)m_prs->m_ppvLengths[nField];
        for (int nRow = 0; nRow < nRowsFetched; nRow++)
        {
            int nStatus = m_prs->GetRowStatus(nRow + 1);
            // Формирование отображаемой строки
            if (nStatus == SQL_ROW_DELETED)
                strData = _T("<DELETED>");
            else if (nStatus == SQL_ROW_NOROW)
                // Такого статуса быть не должно: есть проверка длины выборки
                strData = _T("<NO_ROW>");
            else if (rgLength[nRow] == SQL_NULL_DATA)
                strData = _T("<NULL>");
            else
                strData = &rgData[nRow * MAX_TEXT_LEN];
            // Добавление значений полей в список
            if (nField == 0)
                m_listData.InsertItem(nRow, strData);
            else
            {
                m_listData.SetItem(nRow, nField, LVIF_TEXT,
                                   strData, -1, 0, 0, 0);
            }
        }
    }
}

void CDataDialog::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
}

```

```

        // Обмен данными между элементом управления список и
        // переменной класса CListCtrl
        DDX_Control(pDX, IDC_DATA_LIST, m_listData);
    }
    BEGIN_MESSAGE_MAP(CDataDialog, CDialog)    // Таблица сообщений
        ON_BN_CLICKED(IDC_FIRST, OnFirst)
        ON_BN_CLICKED(IDC_LAST, OnLast)
        ON_BN_CLICKED(IDC_NEXT, OnNext)
        ON_BN_CLICKED(IDC_PREV, OnPrev)
    END_MESSAGE_MAP()

```

Класс CRecordset не поддерживает обновление многострочных наборов данных. Поэтому в производном классе CRecordsetMod используется функция ODBC API SQLSetPos. Класс CRecordsetMod в свою очередь наследуется классом CDynamicSet, используемого для работы с набором данных.

Обновление многострочных данных реализуется следующим образом:

```

CRecordsetMod : public CRecordset
BOOL CRecordsetMod::RowsetUpdate(WORD wRow, WORD wLockType)
{ ASSERT(wRow >= 0 && wRow <= GetRowsetSize());
  RETCODE nRetCode;
  AFX_ODBC_CALL(::SQLSetPos(m_hstmt, wRow,
                           SQL_UPDATE,
                           wLockType));
  return ValidateMod(wRow, SQL_ROW_UPDATED);
}
BOOL CRecordsetMod::RowsetAdd(WORD wRow, WORD wLockType)
{ ASSERT(wRow >= 0 && wRow <= GetRowsetSize() + 1);
  RETCODE nRetCode;
  AFX_ODBC_CALL(::SQLSetPos(m_hstmt, wRow, SQL_ADD, wLockType));
  return ValidateMod(wRow, SQL_ROW_ADDED);
}
BOOL CRecordsetMod::RowsetDelete(WORD wRow, WORD wLockType)
{ ASSERT(wRow >= 0 && wRow <= GetRowsetSize());
  RETCODE nRetCode;

```

```
AFX ODBC CALL(::SQLSetPos(m_hstmt, wRow, SQL_DELETE,
wLockType));
return ValidateMod(wRow, SQL_ROW_DELETED);
}
BOOL CRecordsetMod::ValidateMod(WORD wRow, WORD wExpectedStatus)
{ BOOL bReturn = TRUE;
  if (wRow != 0)
    bReturn = GetRowStatus(wRow) == wExpectedStatus;
  else
  { for (WORD wNum = 1; wNum <= GetRowsetSize(); wNum++)
    {
      // Ошибка, если не определен статус строки
      if (GetRowStatus(wNum) != wExpectedStatus)
        bReturn = FALSE;
    }
  }
  return bReturn;
}
```

## Тема 8. Публикация данных в Интернет.

### **Общие принципы создания серверных приложений**

Данные, отображаемые WEB-браузером, представляют собой HTML-страницу.

Для соединения WEB-браузера и WEB-сервера используется протокол TCP/IP(Transmission Control Protocol/Internet Protocol).

Протокол TCP/IP определяет IP-адрес и номер порта.

IP-адрес задает имя компьютера в сети.

IP-адрес указывается или как числовой идентификатор компьютера или при использовании сервера DNS как символьный псевдоним числового идентификатора.

Локальный компьютер всегда адресуется как 127.0.0.1 или localhost.

При работе в Интернет все используемые IP-адреса уникальны. Поэтому для задания своему ПК некоторого IP-адреса следует получить его у провайдера. При работе без Интернет в локальной сети предприятия можно самостоятельно установить различные IP-адреса для каждого ПК. Например: 192.168.0.2; 192.168.0.3; 192.168.0.4 и т.д.

Номер порта – это значение, однозначно идентифицирующее некоторый логический порт приложения, через который можно получать и посылать данные.

По HTTP-запросу WEB-браузер посылает на WEB-сервер информацию, содержащую URL-адрес документа, тип запроса и значения параметров. URL-адрес может указывать как простую HTML-страницу, так и приложение, выполняемое на WEB-сервере. Такое приложение иногда называется *серверным приложением*.

К серверным приложениям относятся *CGI-приложения* и *ISAPI-приложения*. В отличие от CGI-приложений, выполняемых в отдельном процессе, ISAPI-приложения реализуются как DLL-библиотеки.

Результатом выполнения CGI или ISAPI приложения чаще всего является динамически сформированная HTML-страница.

HTTP-запрос формируется в соответствии с протоколом HTTP (HyperText Transfer Protocol). В HTTP-запросе указывает GET или POST метод передачи данных. При вводе URL-адреса в поле адреса WEB-браузера или выполнении формы (пользователь щелкнул по кнопке типа SUBMIT) выполняется переход по соответствующему URL-адресу.

Если форма содержала данные, то они будут добавлены в конец строки с URL-адресом ( для GET-метода). Соответственно такой способ накладывает ограничение на размер передаваемых параметров. Если при выполнении формы атрибут METHOD установлен равным POST, то используется POST-метод, при котором сначала на сервер посылается строка POST-запроса и HTTP-заголовки запроса, а затем пустая строка и строка, содержащая передаваемые параметры.

ISAPI-приложение может использоваться как для формирования динамической HTML-страницы, так и для обработки параметров, получаемых от формы.

В строке адреса WEB-броузера может быть указан только непосредственно URL-адрес выполняемого ISAPI-приложения. В этом случае для обработки такого запроса используется команда, определяемая макросом `DEFAULT_PARSE_COMMAND`.

Если после URL-адреса за вопросительным знаком следует идентификатор команды, то для обработки данного HTTP-запроса следует использовать функцию сопоставленную данной команде макросом `ON_PARSE_COMMAND`.

Например, для обработки строки

`http://LOCALHOST/MyISAPI_1.dll?Myfunc?string1&123`

ISAPI-приложение будет вызывать функцию `Myfunc` и передавать ей в качестве параметров три значения: первым параметром всегда будет указатель на HTTP-контекст, а далее будут следовать параметры, указанные в HTTP-запросе. В данном примере это два параметра `string1` и `123`. Между собой параметры разделяются символом `&`.

### **ISAPI-приложения**

Для того чтобы создать ISAPI-приложение, иногда также называемое *ISAPI-расширением*, создайте новый проект и в диалоге New Project выберите шаблон создаваемого документа MFC ISAPI Extension Dll.

Далее перейдите на страницу Object Setting мастера ISAPI Extension Wizard и установите флажок Generate a server extension object.

В результате выполненных действий мастер ISAPI Extension Wizard сформирует шаблон ISAPI-приложения. В заголовочном файле `StdAfx.h` добавляемом в шаблон любого приложения, использующего MFC-библиотеку, указана строка `#include <afxisapi.h>`, выполняющая подключение файла `afxisapi.h`, объявляющего следующие классы, поддерживающие работу с HTTP-запросами:

```
class CHtmlStream;
```

```

class CHttpServerContext;
class CHttpServer;
class CHttpFilterContext;
class CHttpFilter;
class CHttpArgList.

```

ISAPI-приложение создается как DLL-библиотека. Класс, выполняющий обработку HTTP-запроса формирующий динамическую HTML-страницу, наследуется от класса CHttpServer.

В следующем листинге приведен код заголовочного файла и файла реализации класса наследуемого от CHttpServer. В автоматически сформированный код ISAPI-приложения внесены изменения, добавляющие в создаваемую HTML-страницу две строки текста и форму, содержащую элемент управления.

*Листинг:*

```

// Заголовочный файл MyISAPI_1.h
#pragma once
#include "resource.h"
class CMyISAPI_1Extension : public CHttpServer
{
public:
    CMyISAPI_1Extension();    // Конструктор
    ~CMyISAPI_1Extension();
public:
    virtual BOOL GetExtensionVersion(HSE_VERSION_INFO* pVer);
    virtual BOOL TerminateExtension(DWORD dwFlags);
    void Default(CHttpServerContext* pCtxt);
    DECLARE_PARSE_MAP()
};
// Файл реализации MyISAPI_1.cpp
#include "stdafx.h"
#include "MyISAPI_1.h"
CWinApp theApp;    // Объект приложение
BEGIN_PARSE_MAP(CMyISAPI_1Extension, CHttpServer)    // Таблица

```

```

// обработки команды
// TODO: место для определения ON_PARSE_COMMAND() и
// ON_PARSE_COMMAND_PARAMS()
ON_PARSE_COMMAND(Default, CMyISAPI_1Extension, ITS_EMPTY)
DEFAULT_PARSE_COMMAND(Default, CMyISAPI_1Extension)
END_PARSE_MAP(CMyISAPI_1Extension)
CMyISAPI_1Extension theExtension; // Только один объект
//ISAPI-расширение класса,
// наследуемого от CHttpServer
CMyISAPI_1Extension::CMyISAPI_1Extension() // Конструктор
{
}
CMyISAPI_1Extension::~CMyISAPI_1Extension()
{
}
BOOL CMyISAPI_1Extension::GetExtensionVersion(
    HSE_VERSION_INFO* pVer)
{
    // Вызов метода базового класса
    CHttpServer::GetExtensionVersion(pVer);
    // Загрузка строки описания
    TCHAR sz[HSE_MAX_EXT_DLL_NAME_LEN+1];
    ISAPIVERIFY(::LoadString(AfxGetResourceHandle(),
        IDS_SERVER, sz, HSE_MAX_EXT_DLL_NAME_LEN));
    _tcscpy(pVer->lpszExtensionDesc, sz);
    return TRUE;
}
BOOL CMyISAPI_1Extension::TerminateExtension(DWORD dwFlags)
{
    // Завершение работы ISAPI-расширения
    //TODO: Clean up any per-instance resources
    return TRUE;
}
// CMyISAPI_1Extension : методы обработчики
// Код формируемой HTML-страницы записывается методом Default
// в поток вывода
void CMyISAPI_1Extension::Default(CHttpServerContext* pCtxt)
{
    StartContent(pCtxt); // Начало HTML-страницы
    WriteTitle(pCtxt); // Формирование значения тега TITLE
    *pCtxt << _T(" HTML-page from "); // Первая строка

```

```

// HTML-страницы
*pCtxt << _T("ISAPI-application ");
// Формирование строки HTML-документа для отображения формы
*pCtxt << _T("<FORM> <INPUT TYPE='text' SIZE=20 </FORM>");
EndContent(pCtxt); // Завершение HTML-страницы
}
// Следующие строки вставляются мастером ISAPI Extension Wizard
// только на тот случай, если ISAPI-расширение не будет
// использовать MFC-библиотеку.
// В противном случае эти строки можно удалить
/****
static HINSTANCE g_hInstance;
HINSTANCE AfxISAPI AfxGetResourceHandle()
{ return g_hInstance; }
BOOL WINAPI DllMain(HINSTANCE hInst, ULONG ulReason, // Точка
LPVOID lpReserved) // входа в DLL-модуль
{
    if (ulReason == DLL_PROCESS_ATTACH)
    {
        g_hInstance = hInst;
    }
    return TRUE;
}
****/

```

Для того чтобы добавить функции, используемые для обработки запросов, следует:

1. Создать функцию для каждой команды. При вызове такой функции в качестве параметра ей передается объект типа `CHttpRequestContext`.
2. Указать для каждой команды вход в таблице обработки команд.
3. При необходимости для реализации собственной обработки запроса надо переопределить метод `CHttpExtensionProc`.

### Таблица описания команд

Для таблицы описания команд в MFC-библиотеку включены пять следующих макросов:



- **BEGIN\_PARSE\_MAP** – определяет начало таблицы описания команд и указывает класс функций членов и базовый класс;
- **END\_PARSE\_MAP** - определяет конец таблицы описания команд;
- **ON\_PARSE\_COMMAND** – идентифицирует команду и указывает соответствующую ей функцию;
- **ON\_PARSE\_COMMAND\_PARAMS** – определяет список параметров обрабатываемой команды. Этот макрос должен следовать непосредственно за макросом **ON\_PARSE\_COMMAND**;
- **DEFAULT\_PARSE\_COMMAND** – определяет команду, используемую в том случае, если нет явного указания выполняемой команды.

Макрос **ON\_PARSE\_COMMAND** используется при определении команды для объекта класса **CHttpServer** (или наследуемого от него), поступающей от клиента, и имеет следующее описание:

**ON\_PARSE\_COMMAND** (*FnName*, *mapClass*, *Args*)

Параметры:

*FnName* – имя функции члена класса, а также и имя команды.

*mapClass* – имя класса указанной функции.

*Args* – указывает тип списка параметров и может принимать следующие значения:

- |                  |                                                                                                                                                        |
|------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>ITS_EMPTY</b> | - параметров нет;                                                                                                                                      |
| <b>ITS_PSTR</b>  | - указатель на строку;                                                                                                                                 |
| <b>ITS_RAW</b>   | - данные предварительно необрабатываемые.<br>Используется в том случае, если список параметров<br>HTTP-запроса может иметь различное число параметров; |
| <b>ITS_I2</b>    | - значение типа short                                                                                                                                  |
| <b>ITS_I4</b>    | - значение типа long                                                                                                                                   |
| <b>ITS_R4</b>    | - значение типа float                                                                                                                                  |
| <b>ITS_R8</b>    | - значение типа double                                                                                                                                 |

ITS\_I8 - значение типа 64-битовое integer

ITS\_ARGLIST - указатель на объект типа CHttpArgList.

Например:

```
BEGIN_PARSE_MAP(CDerivedClass, CHttpServer)
    DEFAULT_PARSE_COMMAND(Myfunc, CDerivedClass)
        // Для запроса типа
        // http://LOCALSERVER/MyISAPI_1.dll?Myfunc&string1&135
    ON_PARSE_COMMAND(Myfunc, CDerivedClass, ITS_PSTR ITS_I2)
    ON_PARSE_COMMAND_PARAMS("string integer=42")

    // Для запроса с тремя параметрами
    ON_PARSE_COMMAND(Myfunc2, CDerivedClass, ITS_PSTR
        ITS_I2 ITS_PSTR)
    ON_PARSE_COMMAND_PARAMS("string integer string2='Default
value'")

    DEFAULT_PARSE_COMMAND(Myfunc3, CDerivedClass)
    ON_PARSE_COMMAND(Myfunc3, CDerivedClass, ITS_RAW)
END_PARSE_MAP(CDerivedClass)

// Функции, выполняемые для обработки команд
void Myfunc(CHttpServerContext* pCtxt, LPTSTR pszName, int
nNumber)
{    }    // Первый параметр стандартен для всех функций,
           // обрабатывающих команды, тип второго и третьего
           // параметра был указан в макросе ON_PARSE_COMMAND
void Myfunc2(CHttpServerContext* pCtxt, LPTSTR pszName,
            int nNumber, LPTSTR pszTitle)
{    }
```

```

void CDerivedClass::Myfunc(CHttpServerContext* pCtxt, void*
pVoid, DWORD dwBytes)      // Используется тип параметров ITS_RAW
{    }                      // pVoid - указатель на передаваемые данные
                           // dwBytes - количество переданных байтов данных

```

Для запросов типа

`http://LOCALSERVER/MyISAPI_2.dll?Myfunc&s1=10&s2=35&c1=y`

можно использовать макрос с типом параметров ITS\_ARGLIST:

`ON_PARSE_COMMAND(Myfunc, CMyHttpServer, ITS_ARGLIST)`. Далее

для разбора такого списка параметром используется класс `CHttpArgList`.

Класс `CHttpArgList` представляет собой массив структур типа `CHttpArg`.

При этом данные доступны через объект `CHttpArg`.

Поле `CHttpArg::m_pstrValue` содержит значение параметра,

а поле `CHttpArg::m_pstrArg` – имя параметра.

Например, для строки

`http://localhost/myh1.dll&Arg1=str 1&Arg2&Arg3=str 2`

надо реализовать разбор параметров по следующей схеме:

`CHttpArgList`

```

| GetFirstArg()
|----->
| GetNextArg()
|
|
| GetNextArg()
|----->
|
|
|

```

**CHttpArg**

```

m_pstrArg="Arg1"
m_pstrValue="str 1"
m_pstrRaw="Arg1 = str 1"

```

**CHttpArg**

```

m_pstrArg="Arg2"
m_pstrValue=""
m_pstrRaw="Arg2"

```

| GetNextArg()  
|----->

**CHttpArg**

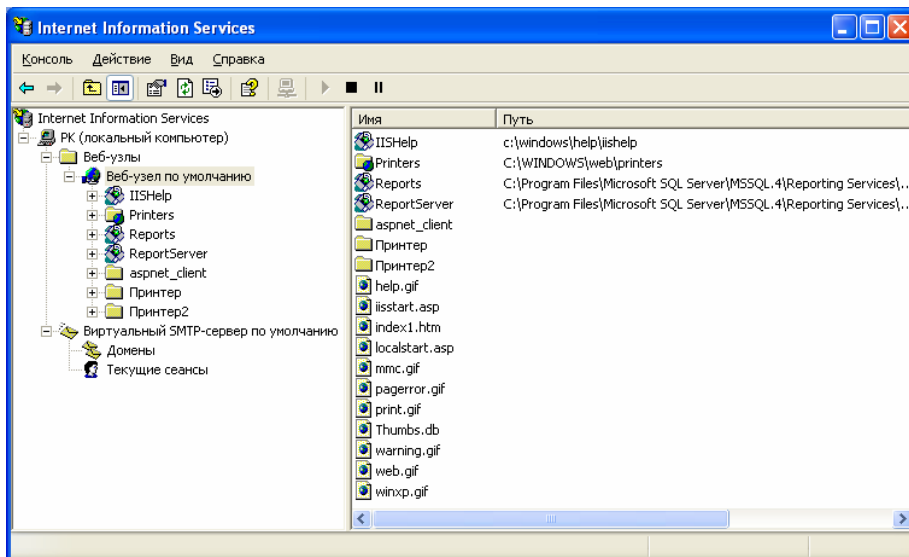
m\_pstrArg="Arg3"

m\_pstrValue="str 3"

m\_pstrRaw="Arg3 = str 3"

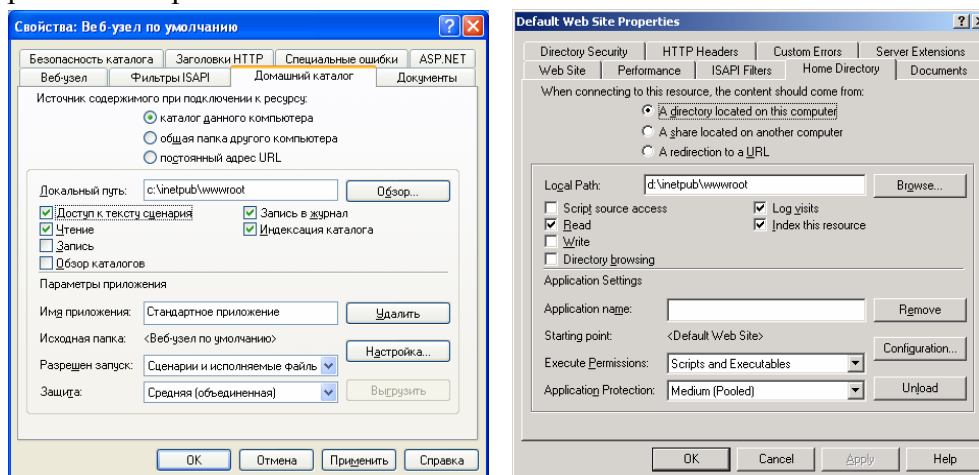
Для выполнения ISAPI-приложения соответствующую DLL-библиотеку следует поместить в каталог WEB-сервера. Таким сервером может быть Personal Web Server, Internet Information Server или любой другой WEB-сервер.

Для просмотра сведений о виртуальных каталогах WEB-сервера следует открыть диалог для администрирования WEB-сервера (для IIS сервера следует выбрать пиктограмму Internet Services Manager). Отображаемый далее диалог Internet Information Services позволяет получать информацию и настраивать ISS сервер.



Для того чтобы получить информацию о расположении домашнего каталога ISS сервера, следует на панели расположенной слева выделить элемент Веб-узел по умолчанию (Default Web Site) и выполнить команду контекстного меню Свойства (Properties).

Далее на странице Домашний каталог (Home Directory) в поле Локальный путь (Local Path) указано расположение домашнего каталога. Для того чтобы из данного каталога можно было загружать как HTML-файлы, так и DLL-файлы, значение поля Executable Permissions должно быть установлено равным Script and Executables.

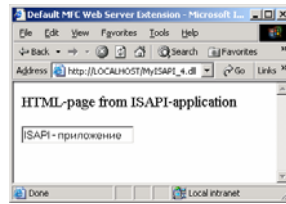


После размещения ISAPI-приложения в домашнем каталоге WEB-сервера, это приложение можно выполнить в WEB-браузере, указав соответствующий URL-адрес.

При выполнении приложения на локальном компьютере в качестве имени сервера указывается LOCALHOST.

Например: [http://LOCALHOST/MyISAPI\\_4.dll](http://LOCALHOST/MyISAPI_4.dll).

При размещении ISAPI-приложения на WEB-сервере имя приложения можно изменить. На следующем рисунке приведен результат выполнения в WEB-браузере описанного выше ISAPI-приложения.



При первом выполнении ISAPI-приложения сервер загружает данную DLL. При всех последующих вызовах обращение происходит к уже загруженной DLL.

## WEB в .NET

Web-формы можно добавлять в следующие проекты .NET:

- ASP.NET Web Site
- ASP.NET Web Service
- Empty Web Site
- Personal Web Site Starter Kit

Расположить проект можно как в файловой системе, так и на Web-сервере. Если выбрано расположение HTTP, то для локального сервера поле Location следует установить равным `http://localhost/каталог_проекта`.

Файлы Web-приложения:

- **Default.aspx** – текст HTML-страницы;
- **Default.aspx.cs** – подключаемые пространства имен и метод `Page_Load`.

Если добавить новый элемент "таблицу стилей", то в проект добавляется файл **StyleSheet.css**.

Следующий пример иллюстрирует создание простого ASP-приложения.

```
// Default.aspx.cs
using System;
using System.Data;
```

```
using System.Configuration;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Web.UI.HtmlControls;

public partial class _Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
    }
}

// Default.aspx
<%@ Page Language="C#" AutoEventWireup="true"
CodeFile="Default.aspx.cs" Inherits="_Default" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0
Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>Untitled Page</title>
    <link rel="stylesheet" type="text/css"
        href="StyleSheet.css" />
</head>
<script language="javascript" type="text/javascript">
// <![CDATA[
```

```
function Submit1_onclick() {
alert("1234567");
}

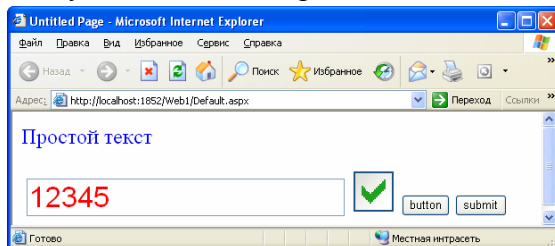
// ]]>
</script>
</head>
<body>
  <form id="form1" runat="server">
    <div title="Заголовок">
      &nbsp;<input id="Text1" type="text"
        class="C11" value="12345" />
      <input id="Checkbox1" checked="checked"
        style="width: 52px; height: 56px"
        title="Флажок1"
        type="checkbox" />
      <input id="Button1" title="Кнопка 1"
        type="button" value="button" />
      <input id="Submit1" type="submit" value="submit"
        onclick="return Submit1_onclick()" />
    </div>
  </form>
</body>
</html>

// StyleSheet.css
body
{
  font-size: 18pt;
```



```
        color: blue;
    }
    .C11
    {
        font-size: 24pt;
        color: red;
    }
}
```

В результате выполнения данного серверного приложения будет отображена следующая HTML-страница.



## Тема 9. Применение средств ASP.NET для доступа к данным

Для реализации доступа к данным ASP.NET позволяет применять как классы пространств имен System.Data и System.Xml, так и модель декларативного связывания данных. Этот механизм является более новым и основан на использовании двух типов серверных элементов управления : источников данных (data source controls) и элементов, связанных с данными (data-bound controls).

Элементы управления источники данных применяются для реализации доступа к данным (соединение, чтение, запись данных), а элементы управления, связанные с данными, используются для визуализации данных. ASP.NET предоставляет следующие элементы управления источники данных:

- `ObjectDataSource` используется в Web-приложении для управления данными;
- `SqlDataSource` - позволяет использовать провайдеры данных ADO.NET;
- `AccessDataSource` позволяет работать с базой данных Microsoft Access;
- `XmlDataSource` позволяет работать с источником данных – XML-файлом;
- `SiteMapDataSource` - применяется с навигацией ASP.NET.

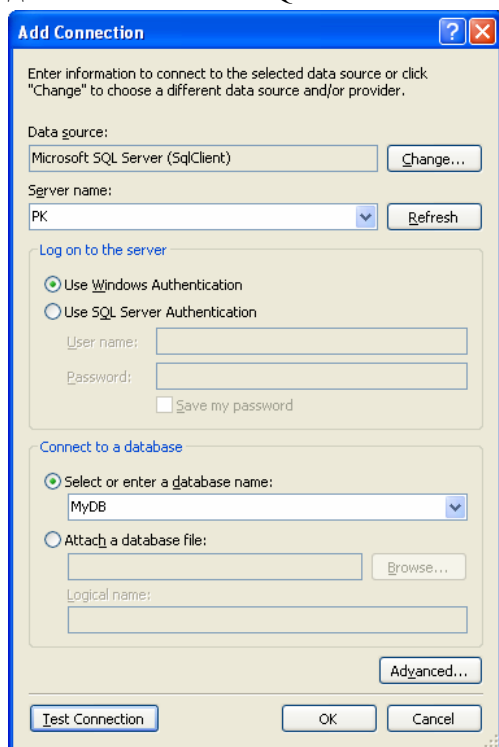
ASP.NET предоставляет следующие элементы управления, связанные с данными:

- `ListBox`, `DropDownList`, `BulletedList`, `CheckBoxList`, `RadioButtonList` – отображение данных в виде списков;
- `AdRotator` – отображение изображений, используемых как гиперссылка;
- `DataList` – отображение данных в виде таблицы;
- `DetailsView` – представляет данные одной строки в виде таблицы (каждое поле отдельной строкой);
- `FormView` – представляет данные одной строки, но в отличие от `DetailsView`, позволяет компоновку каждой записи;
- `GridView` – отображение данных в виде таблицы, позволяя выполнять редактирование, изменение и сортировку данных без какого-либо программирования. Этот элемент управления заменяет в версии 2.0 элемент управления `DataGrid`;
- `Menu` – отображает данные в виде иерархического динамического меню, возможно включающего подменю;
- `Repeater` – отображает данные в виде списка, в котором каждый отображаемый элемент использует определяемый разработчиком шаблон элемента;
- `TreeView` – отображает данные в виде иерархического дерева.

Для того чтобы создать Web-приложение использующее для доступа к данным ASP.NET, следует выбрать соответствующий шаблон, а затем в окне дизайнера добавить элемент управления источник данных со страницы Data палитры компонентов.

Для примера добавим компонент SqlDataSource. После добавления в окне дизайнера данного компонента выберем для него команду Configure Data Source. Это позволит создать строку соединения, сохранить ее и определить извлекаемый набор данных.

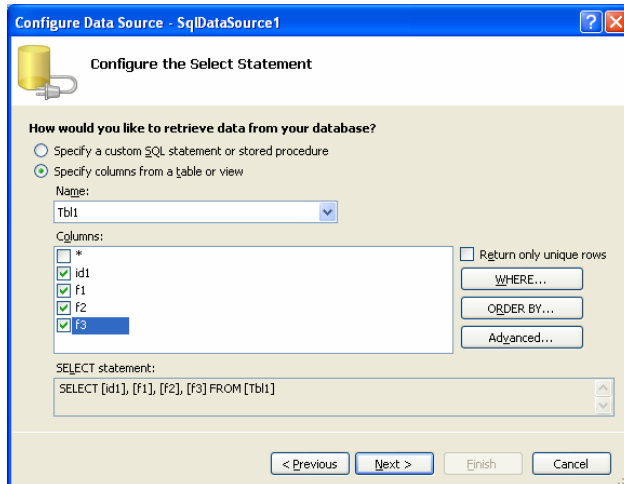
Следующий диалог иллюстрирует выбор в качестве источника данных базы данных Microsoft SQL Server.



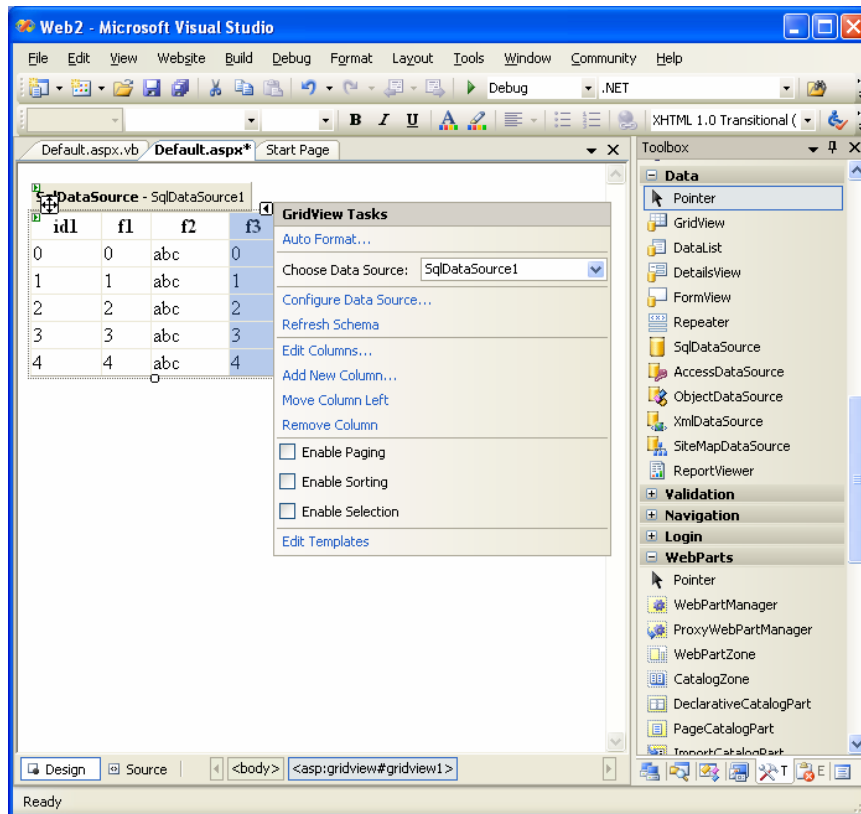
В результате будет сформирована следующая строка соединения:

Data Source=PK;Initial Catalog=MyDB;Integrated Security=True

Далее следует выбрать данные, которые будут извлечены в набор данных.



Для отображения данных из сформированного источника данных в виде таблицы следует добавить в окне дизайнера компонент GridView со страницы Data палитры компонентов и установит связь с созданным источником данных, выбрав команду Choose Data Source.



В результате будет автоматически сформирован следующий код:

```
<%@ Page Language="VB" AutoEventWireup="false"
CodeFile="Default.aspx.vb" Inherits="_Default" %>

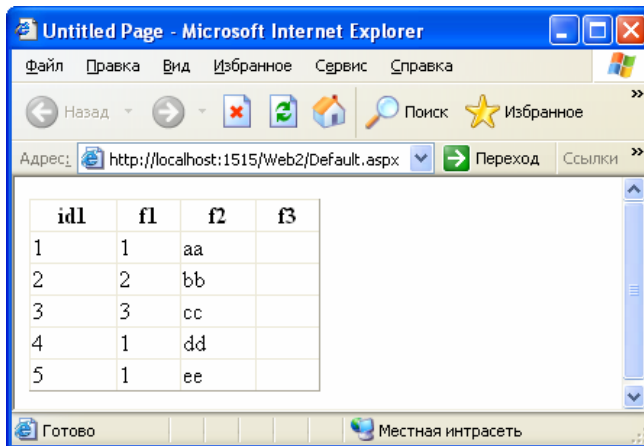
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>Untitled Page</title>
```

```
</head>
<body>
  <form id="form1" runat="server">
    <div>
      <asp:SqlDataSource ID="SqlDataSource1" runat="server"
ConnectionString="<%$ ConnectionStrings:MyDBConnectionString %>"
SelectCommand="SELECT [id1], [f1], [f2], [f3] FROM
[Tbl1]"></asp:SqlDataSource>

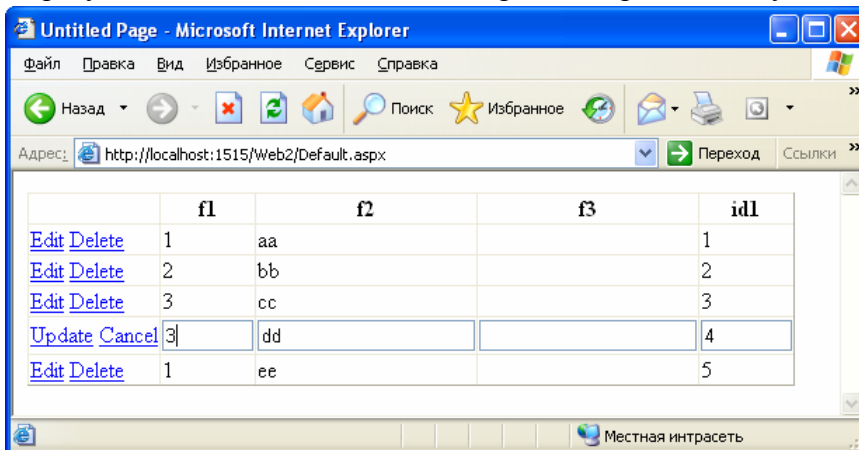
    </div>
    <asp:GridView ID="GridView1" runat="server"
AutoGenerateColumns="False" DataSourceID="SqlDataSource1"
Width="201px">
      <Columns>
        <asp:BoundField DataField="id1" HeaderText="id1"
InsertVisible="False" ReadOnly="True"
SortExpression="id1" />
        <asp:BoundField DataField="f1" HeaderText="f1"
SortExpression="f1" />
        <asp:BoundField DataField="f2" HeaderText="f2"
SortExpression="f2" />
        <asp:BoundField DataField="f3" HeaderText="f3"
SortExpression="f3" />
      </Columns>
    </asp:GridView>
  </form>
</body>
</html>
```

При выполнении данного серверного приложения будет отображена следующая HTML-страница.



Для того чтобы разместить кнопки редактирования, следует для объекта GridView установить значения свойств `AutoGenerateEditButton` и `AutoGenerateDeleteButton` равными `True`.

В результате создаваемая HTML-страница примет следующий вид.



При этом будет автоматически сформирован следующий код:

```
<body>  
  <form id="form1" runat="server">
```

```

<div>
  <asp:SqlDataSource ID="SqlDataSource1" runat="server"
    ConnectionString="<%$ ConnectionStrings:MyDBConnectionString %>"
    SelectCommand="SELECT [id1], [f1], [f2], [f3] FROM [Tbl1]">
  </asp:SqlDataSource>
</div>
<asp:GridView ID="GridView1" runat="server"
  AutoGenerateColumns="False" DataSourceID="SqlDataSource1"
  Width="201px" AutoGenerateDeleteButton="True"
  AutoGenerateEditButton="True">
  <Columns>
    <asp:BoundField DataField="f1" HeaderText="f1"
      SortExpression="f1" />
    <asp:BoundField DataField="f2" HeaderText="f2"
      SortExpression="f2" />
    <asp:BoundField DataField="f3" HeaderText="f3"
      SortExpression="f3" />
    <asp:BoundField DataField="id1" HeaderText="id1"
      InsertVisible="False" SortExpression="id1" />
  </Columns>
</asp:GridView>
</form>
</body>

```

Элемент управления GridView имеет следующее формальное описание:

```

<asp:GridView
  AccessKey="string"
  AllowPaging="True|False"
  AllowSorting="True|False"
  AutoGenerateColumns="True|False"
  AutoGenerateDeleteButton="True|False"
  AutoGenerateEditButton="True|False"
  AutoGenerateSelectButton="True|False"
  BackColor="color name|#dddddd"
  BackImageUrl="uri"
  BorderColor="color name|#dddddd"
  BorderStyle="NotSet|None|Dotted|Dashed|Solid|Double|Groove|
    Ridge|Inset|Outset"

```



```

BorderWidth="size"
Caption="string"
CaptionAlign="NotSet|Top|Bottom|Left|Right"
CellPadding="integer"
CellSpacing="integer"
CssClass="string"
DataKeyNames="string"
DataMember="string"
DataSource="string"
DataSourceID="string"
EditIndex="integer"
EmptyDataText="string"
Enabled="True|False"
EnableSortingAndPagingCallbacks="True|False"
EnableTheming="True|False"
EnableViewState="True|False"
Font-Bold="True|False"
Font-Italic="True|False"
Font-Names="string"
Font-Overline="True|False"
Font-Size="string|Smaller|Larger|XX-Small|
    X-Small|Small|Medium|Large|X-Large|XX-Large"
Font-Strikeout="True|False"
Font-Underline="True|False"
ForeColor="color name|#ddddd"
GridLines="None|Horizontal|Vertical|Both"
Height="size"
HorizontalAlign="NotSet|Left|Center|Right|Justify"
ID="string"
OnDataBinding="DataBinding обработчик события "
OnDataBound="DataBound обработчик события "
OnDisposed="Disposed обработчик события "
OnInit="Init обработчик события "
OnLoad="Load обработчик события "
OnPageIndexChanged="PageIndexChanged обработчик события "
OnPageIndexChanging="PageIndexChanging обработчик события "
OnPreRender="PreRender обработчик события "
OnRowCancelingEdit="RowCancelingEdit обработчик события "
OnRowCommand="RowCommand обработчик события "
OnRowCreated="RowCreated обработчик события "
OnRowDataBound="RowDataBound обработчик события "

```

```

OnRowDeleted="RowDeleted обработчик события "
OnRowDeleting="RowDeleting обработчик события "
OnRowEditing="RowEditing обработчик события "
OnRowUpdated="RowUpdated обработчик события "
OnRowUpdating="RowUpdating обработчик события "
OnSelectedIndexChanged="SelectedIndexChanged обработчик
    события"
OnSelectedIndexChanging="SelectedIndexChanging обработчик
    события"
OnSorted="Sorted обработчик события "
OnSorting="Sorting обработчик события "
OnUnload="Unload обработчик события "
PageIndex="integer"
PagerSettings-FirstPageImageUrl="uri"
PagerSettings-FirstPageText="string"
PagerSettings-LastPageImageUrl="uri"
PagerSettings-LastPageText="string"
PagerSettings-Mode=
    "NextPrevious|Numeric|NextPreviousFirstLast|NumericFirstLast"
PagerSettings-NextPageImageUrl="uri"
PagerSettings-NextPageText="string"
PagerSettings-PageButtonCount="integer"
PagerSettings-Position="Bottom|Top|TopAndBottom"
PagerSettings-PreviousPageImageUrl="uri"
PagerSettings-PreviousPageText="string"
PagerSettings-Visible="True|False"
PageSize="integer"
RowHeaderColumn="string"
runat="server"
SelectedIndex="integer"
ShowFooter="True|False"
ShowHeader="True|False"
SkinID="string"
Style="string"
TabIndex="integer"
ToolTip="string"
UseAccessibleHeader="True|False"
Visible="True|False"
Width="size"
>
<AlternatingRowStyle />

```

```
<Columns>
  <asp:BoundField
    AccessibleHeaderText="string"
    ApplyFormatInEditMode="True|False"
    ConvertEmptyStringToNull="True|False"
    DataField="string"
    DataFormatString="string"
    FooterText="string"
    HeaderImageUrl="uri"
    HeaderText="string"
    HtmlEncode="True|False"
    InsertVisible="True|False"
    NullDisplayText="string"
    ReadOnly="True|False"
    ShowHeader="True|False"
    SortExpression="string"
    Visible="True|False"
  >
    <ControlStyle />
    <FooterStyle />
    <HeaderStyle />
    <ItemStyle />
  </asp:BoundField>
  <asp:ButtonField
    AccessibleHeaderText="string"
    ButtonType="Button|Image|Link"
    CausesValidation="True|False"
    CommandName="string"
    DataTextField="string"
    DataTextFormatString="string"
    FooterText="string"
    HeaderImageUrl="uri"
    HeaderText="string"
    ImageUrl="uri"
    InsertVisible="True|False"
    ShowHeader="True|False"
    SortExpression="string"
    Text="string"
    ValidationGroup="string"
    Visible="True|False"
  >
```

```
<ControlStyle />
<FooterStyle />
<HeaderStyle />
<ItemStyle />
</asp:ButtonField>
<asp:CheckBoxField
  AccessibleHeaderText="string"
  DataField="string"
  FooterText="string"
  HeaderImageUrl="uri"
  HeaderText="string"
  InsertVisible="True|False"
  ReadOnly="True|False"
  ShowHeader="True|False"
  SortExpression="string"
  Text="string"
  Visible="True|False"
>

  <ControlStyle />
  <FooterStyle />
  <HeaderStyle />
  <ItemStyle />
</asp:CheckBoxField>
<asp:CommandField
  AccessibleHeaderText="string"
  ButtonType="Button|Image|Link"
  CancelImageUrl="uri"
  CancelText="string"
  CausesValidation="True|False"
  DeleteImageUrl="uri"
  DeleteText="string"
  EditImageUrl="uri"
  EditText="string"
  FooterText="string"
  HeaderImageUrl="uri"
  HeaderText="string"
  InsertImageUrl="uri"
  InsertText="string"
  InsertVisible="True|False"
  NewImageUrl="uri"
  NewText="string"
```

```

        SelectImageUrl="uri"
        SelectText="string"
        ShowCancelButton="True|False"
        ShowDeleteButton="True|False"
        ShowEditButton="True|False"
        ShowHeader="True|False"
        ShowInsertButton="True|False"
        ShowSelectButton="True|False"
        SortExpression="string"
        UpdateImageUrl="uri"
        UpdateText="string"
        ValidationGroup="string"
        Visible="True|False"
>
        <ControlStyle />
        <FooterStyle />
        <HeaderStyle />
        <ItemStyle />
</asp:CommandField>
<asp:HyperLinkField
    AccessibleHeaderText="string"
    DataNavigateUrlFields="string"
    DataNavigateUrlFormatString="string"
    DataTextField="string"
    DataTextFormatString="string"
    FooterText="string"
    HeaderImageUrl="uri"
    HeaderText="string"
    InsertVisible="True|False"
    NavigateUrl="uri"
    ShowHeader="True|False"
    SortExpression="string"
    Target="string|_blank|_parent|_search|_self|
        _top"
    Text="string"
    Visible="True|False"
>
        <ControlStyle />
        <FooterStyle />
        <HeaderStyle />
        <ItemStyle />

```

```

</asp:HyperLinkField>
<asp:ImageField
  AccessibleHeaderText="string"
  AlternateText="string"
  ConvertEmptyStringToNull="True|False"
  DataAlternateTextField="string"
  DataAlternateTextFormatString="string"
  DataImageUrlField="string"
  DataImageUrlFormatString="string"
  FooterText="string"
  HeaderImageUrl="uri"
  HeaderText="string"
  InsertVisible="True|False"
  NullDisplayText="string"
  NullImageUrl="uri"
  ReadOnly="True|False"
  ShowHeader="True|False"
  SortExpression="string"
  Visible="True|False"
>
    <ControlStyle />
    <FooterStyle />
    <HeaderStyle />
    <ItemStyle />
</asp:ImageField>
<asp:TemplateField
  AccessibleHeaderText="string"
  ConvertEmptyStringToNull="True|False"
  FooterText="string"
  HeaderImageUrl="uri"
  HeaderText="string"
  InsertVisible="True|False"
  ShowHeader="True|False"
  SortExpression="string"
  Visible="True|False"
>
    <ControlStyle />
    <FooterStyle />
    <HeaderStyle />
    <ItemStyle />
    <AlternatingItemTemplate>

```

```

        <!-- child controls -->
    </AlternatingItemTemplate>
    <EditItemTemplate>
        <!-- child controls -->
    </EditItemTemplate>
    <FooterTemplate>
        <!-- child controls -->
    </FooterTemplate>
    <HeaderTemplate>
        <!-- child controls -->
    </HeaderTemplate>
    <InsertItemTemplate>
        <!-- child controls -->
    </InsertItemTemplate>
    <ItemTemplate>
        <!-- child controls -->
    </ItemTemplate>
</asp:TemplateField>
</Columns>
<EditRowStyle />
<EmptyDataRowStyle />
<EmptyDataTemplate>
    <!-- child controls -->
</EmptyDataTemplate>
<FooterStyle />
<HeaderStyle />
<PagerSettings
    FirstPageImageUrl="uri"
    FirstPageText="string"
    LastPageImageUrl="uri"
    LastPageText="string"
    Mode="NextPrevious|Numeric|NextPreviousFirstLast|
        NumericFirstLast"
    NextPageImageUrl="uri"
    NextPageText="string"
    OnPropertyChanged="PropertyChanged event handler"
    PageButtonCount="integer"
    Position="Bottom|Top|TopAndBottom"
    PreviousPageImageUrl="uri"
    PreviousPageText="string"
    Visible="True|False"

```

```
        />
        <PagerStyle />
        <PagerTemplate>
            <!-- child controls -->
        </PagerTemplate>
        <RowStyle />
        <SelectedRowStyle />
    </asp:GridView>
```

Строки, отображаемые компонентом GridView можно редактировать программным путем, используя метод UpdateRow. Следующий пример иллюстрирует использование данного метода.

```
<%@ Page language="C#" %>
<script runat="server">
    void UpdateRowButton_Click(Object sender,
                                EventArgs e)
    {
        // Вызов метода для текущей записи
        // в режиме редактирования
        GridView1.UpdateRow(GridView1.EditIndex, true);
    }

    void GridView1_RowCommand(Object sender,
                                GridViewCommandEventArgs e)
    {
        // Делаем кнопку UpdateRowButton доступной только,
        // если компонент GridView находится
        // в режиме редактирования
        switch (e.CommandName)
        { case "Edit":
            UpdateRowButton.Enabled = true;
            break;
          case "Cancel":
            UpdateRowButton.Enabled = false;
```



```
        break;
    case "Update":
        UpdateRowButton.Enabled = false;
        break;
    default:
        UpdateRowButton.Enabled = false;
        break;
    }
}

</script>

<html>
<body>
    <form runat="server">
        <asp:button id="UpdateRowButton"
            text="Записать строку"
            enabled="false"
            onclick="UpdateRowButton_Click"
            runat="server"/>

        <hr/>
        <!-- Компонент GridView автоматически установит столбцы -->
        <!-- определенные свойством datakeynames как только-для-чтения. -->
        <!-- Для этих столбцов не будет отображено полей ввода -->
        <!-- в режиме редактирования. -->
        <asp:gridview id="GridView1"
            allowpaging="true"
            datasourceid="SqlDataSource1"
            autogeneratecolumns="true"
            autogenerateeditbutton="true"
            datakeynames="idl"
            onrowcommand="GridView1_RowCommand"
            runat="server">
```

```
</asp:gridview>

<!-- Строка соединения расположена в Web.config -->
<asp:sqldatasource id="SqlDataSource1"
    selectcommand="Select [id1], [f1], [f2], [f3]
        From [Tbl1]"
    updatecommand="Update Tbl1 SET f1=@f1, f2=@f2,
        f3=@f3 WHERE (id1 = @id1)"
    connectionstring=
    "<%%$ ConnectionStrings:MyDbConnectionString%"
    runat="server">
</asp:sqldatasource>

</form>
</body>
</html>
```

## Тема 10. Работа с данными в формате XML

### **Объектная модель документа XML**

DOM (Document Object Model) является моделью представления данных, построенной на базе структуры объектов. Все документы в DOM имеют древовидную логическую структуру. Вершиной дерева служит один корневой элемент, который может иметь несколько дочерних элементов.

Если для некоторого документа использовать два DOM-представления, то они всегда создадут одну и ту же структурную модель.

Для определения структуры документа используется язык XML.

XML-документ состоит из дочерних и родительских узлов.

Начальный тег определяет корень документа. Вложенные теги определяют элементы документа. Элементом называется набор тегов и заключенные в

нем данные. Теги могут содержать вместо данных другие теги. Открывающий тег может содержать атрибуты.

XML-файл начинается с пролога, в котором может быть указан номер версии, кодировка, задано определение типа документа DTD (Document Type Defenition). Определение типа документа может быть расположено в самом файле или находиться в отдельном файле. Это указывается в строке пролога DOCTYPE.

Например:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE myschema SYSTEM "xmldtd.dtd">
```

Определение типа документа описывает элементы из которых состоит документ. Узел !DOCTYPE реализуется классом XmlDocumentType.

Например:

```
<!DOCTYPE myschema [
    <!ELEMENT myschema(book)*>
    <!--Каждый элемент book состоит из элементов title, autor и price -->
    <!ELEMENT book (title, autor+, price)>
        <!ELEMENT title (#PCDATA)>
        <!ELEMENT autor (fname, lname)>
            <!ELEMENT fname (#PCDATA)>
            <!ELEMENT lname (#PCDATA)>
        <!ELEMENT price (#PCDATA)>
        <!ATTLIST price type (DOLLAR|EURO|RUBL) #REQUIRED)
]>
```

Суффикс + определяет наличие дочерних элементов (больше или равно 1). Если число элементов равно 0 или 1, то используется суффикс ?, а если число элементов больше или равно 0, то суффикс \*.

Параметр #PCDATA устанавливает, что текст данного элемента не может быть текстом разметки.

Если задать параметр #CDATA, то для этих данных не будет определяться их соответствие определению типа документа.

После пролога располагается тело документа, состоящее из узлов, описывающих инструкции по обработке данных, содержимое, атрибуты, ссылки на объекты, комментарии, фрагменты документов.

Атрибуты могут быть пользовательские и встроенные (определяемые спецификацией языка XML).

Ссылка на объект указывается его именем с префиксом &.

Комментарии указываются внутри тега <!-- -->.

Класс DOM XML представляет XML-документ в памяти, и позволяет читать и записывать документ.

Для доступа к XML-документу можно использовать функции Simple API for XML (SAX). При этом весь XML-документ не загружается в память, а выполняется синтаксический анализ документа, при котором инициируются соответствующие события.

В Framework .NET каждый отдельный узел, представляющий элемент, реализуется классом XmlElement. Сам документ реализуется классом XmlDocument.

### **Класс XmlDocument**

Для загрузки XML-документа достаточно выполнить следующий код:

```
XmlDocument doc= new XmlDocument();  
doc.Load("MyDataFile.xml");  
Console.WriteLine(doc.InnerXml.ToString);
```

Для загрузки XML-документа можно использовать следующие методы;

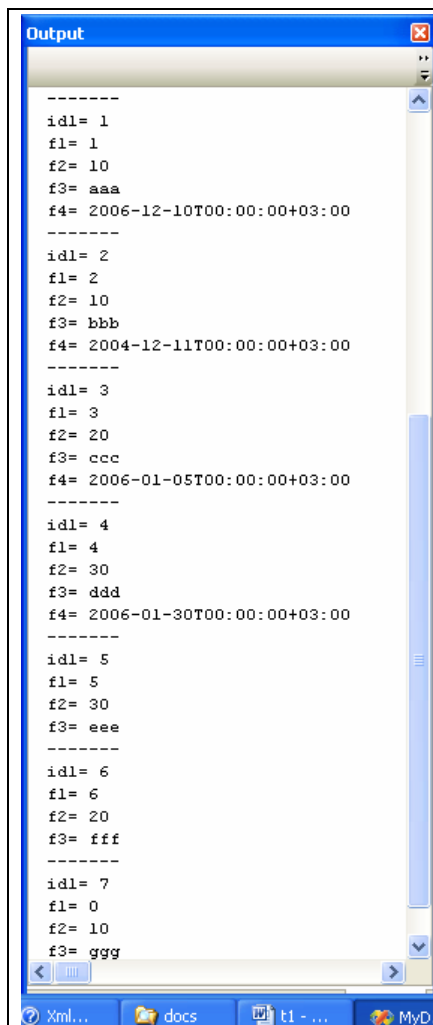
- Load("имя\_xml\_файла")
- LoadXml("содержание\_xml\_документа").

Для сохранения XML-документа можно применять метод Save("имя\_файла") класса XmlDocument.

Каждый узел XML-документа представляется классом XmlNode. Для последовательного доступа ко всем дочерним узлам XML-документа следует использовать свойство ChildNodes класса XmlDocument.

Например:

```
private void button2_Click(object sender, EventArgs e)
{
    mydb1DataSet.Tbl1.WriteXml("MyDataFile.xml");
    XmlDocument doc = new XmlDocument();
    doc.Load("MyDataFile.xml");
    int i, count;
    count = 0; i = 1;
    XmlNode node = doc.ChildNodes[1];
    foreach (XmlNode node1 in node.ChildNodes)
    {
        Console.WriteLine("-----");
        // Отобразим все узлы дочерние к данному
        foreach (XmlNode node2 in node1.ChildNodes)
        {
            Console.WriteLine(doc.DocumentElement.FirstChild.ChildNodes[count]
                .Name + " = " + node2.FirstChild.Value);
            count++;
        }
        i++; count = 0;
    }
}
```



The screenshot shows a window titled "Output" with a text area containing the following text:

```
-----  
id1= 1  
f1= 1  
f2= 10  
f3= aaa  
f4= 2006-12-10T00:00:00+03:00  
-----  
id1= 2  
f1= 2  
f2= 10  
f3= bbb  
f4= 2004-12-11T00:00:00+03:00  
-----  
id1= 3  
f1= 3  
f2= 20  
f3= ccc  
f4= 2006-01-05T00:00:00+03:00  
-----  
id1= 4  
f1= 4  
f2= 30  
f3= ddd  
f4= 2006-01-30T00:00:00+03:00  
-----  
id1= 5  
f1= 5  
f2= 30  
f3= eee  
-----  
id1= 6  
f1= 6  
f2= 20  
f3= fff  
-----  
id1= 7  
f1= 0  
f2= 10  
f3= ggg
```

The window has a standard Windows taskbar at the bottom with icons for "xml...", "docs", "t1 - ...", and "MyD...".

Вывод строк XML-документа при последовательном чтении узлов документа.

### **Классы *XmlReader* и *XmlWriter***

Для реализации однонаправленного чтения XML-документа используются классы, наследуемые от *XmlReader*:

- *XmlTextReader* – позволяет читать XML-документы без их проверки;
- *XmlValidatingReader* – позволяет читать XML-документы и предоставляет доступ к схеме документа.

Класс *XmlTextReader* может читать данные как из XML-файла, так и из потока.

Например:

```
XmlTextReader xml1=new XmlTextReader("MyDataFile.xml");
```

Для реализации записи XML-документа используются классы, наследуемые от *XmlWriter*, например *XmlTextWriter*.

Например:

```
using System;
using System.IO;
using System.Xml;

public class ReadWriteXML
{
    private const string doc = "MyDataFile.xml";
    public static void Main() {
        XmlWriter writer = null;
        try {
            writer = XmlWriter.Create (doc);

            writer.WriteComment("XML документ");

            // Запись корневого элемента
            writer.WriteStartElement("book");
```

```
// Запись атрибута
writer.WriteAttributeString("xmlns",
                           null,
                           "urn:ReadWriteXML");

// Запись элемента title
writer.WriteStartElement("title");
writer.WriteString("Название 1");
writer.WriteEndElement();

// Запись элемента autor
writer.WriteElementString ("autor", "имя 1");

// Запись элемента price.
writer.WriteElementString("price", "135");

// Закрываем тег для корневого элемента
writer.WriteEndElement();

// Запись XML-файла и закрытие потока XML-данных
writer.Flush();
writer.Close();
}
finally {
    if (writer != null) writer.Close();
}
}
```



Метод `WriteStartElement` класса `XmlTextWriter` записывает открывающий тег задаваемого элемента, а метод `WriteEndElement` – записывает закрывающий тег.

Метод `WriteStartDocument` записывает в начало документа строку `<?xml version="1.0" ?>`.

Метод `WriteElementString` записывает символьные данные элемента.