

Министерство образования и науки Российской Федерации
НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

В.М. СТАСЫШИН, Т.Л. СТАСЫШИНА

ТЕХНОЛОГИИ ДОСТУПА К БАЗАМ ДАННЫХ

Утверждено Редакционно-издательским советом университета
в качестве учебного пособия

НОВОСИБИРСК
2014

УДК 004.65(075.8)
С 779

Рецензенты:

зам. директора ЦИУ *О.Т. Аврунев*
канд. техн. наук, доцент каф. ПСиБД *В.Г. Кобылянский*
д-р техн. наук, профессор каф. ПМТ *М.Г. Персова*

Работа подготовлена на кафедре программных систем и баз данных для студентов ФПМИ дневного отделения (направления 010500.62, 010400.62)

Стасьшин В.М.

С 779 Технологии доступа к базам данных: учеб. пособие / В.М. Стасьшин, Т.Л. Стасьшина. – Новосибирск: Изд-во НГТУ, 2014. – 176 с.

ISBN 978-5-7782-2595-4

Рассмотрены различные технологии доступа к базам данных современных СУБД с применением клиентских (ESQL/C, ODBC, CGI, PHP, ADO C++ Builder, Microsoft Excel) и серверных средств (триггеры и процедуры), используемые как для оперативной обработки, так и для анализа данных. Описаны идеология, языковые и программные средства различных способов доступа к базам данных, приведены обширная справочная информация, необходимая для написания программ, и примеры их использования. Включенный в пособие материал входит в программу курсов лекций «Базы данных и экспертные системы», «Технологии баз данных», читаемых студентам факультета прикладной математики и информатики.

Учебное пособие может быть полезно также специалистам, занимающимся информационными технологиями и самостоятельно осваивающим вопросы проектирования и разработки программных приложений над базами данных.

УДК 004.65(075.8)

Стасьшин Владимир Михайлович
Стасьшина Татьяна Леонидовна

ТЕХНОЛОГИИ ДОСТУПА К БАЗАМ ДАННЫХ
Учебное пособие

Редактор *И.Л. Кескевич*
Выпускающий редактор *И.П. Брованова*
Корректор *И.Е. Семенова*
Дизайн обложки *А.В. Ладыжская*
Компьютерная верстка *С.И. Ткачева*

Налоговая льгота – Общероссийский классификатор продукции
Издание соответствует коду 95 3000 ОК 005-93 (ОКП)

Подписано в печать 26.12.2014. Формат 60 × 84 1/16. Бумага офсетная. Тираж 100 экз.
Уч.-изд. л. 10,23. Печ. л. 11,0. Изд. № 218. Заказ № 196. Цена договорная

Отпечатано в типографии
Новосибирского государственного технического университета
630073, г. Новосибирск, пр. К. Маркса, 20

ISBN 978-5-7782-2595-4

© Стасьшин В.М., Стасьшина Т.Л., 2014
© Новосибирский государственный
технический университет, 2014

ОГЛАВЛЕНИЕ

Введение.....	5
1. Встроенный SQL (Embedded ESQL/C).....	8
1.1. Общие принципы работы с SQL	8
1.2. Обработка ошибок.....	11
1.3. Выборка единственной строки.....	13
1.4. Обработка null-значений.....	14
1.5. Обработка нескольких строк.....	16
1.6. Динамический SQL	19
1.7. Примеры использования встроенного SQL	23
2. Доступ к данным на основе стандарта ODBC.....	25
2.1. Стандартизация архитектуры доступа к базе данных.....	25
2.2. Архитектура и уровни соответствия ODBC.....	26
2.3. Функции ODBC и структура команд.....	29
2.4. Подготовительные операции в ODBC-программе	30
2.5. Обработка ошибок в ODBC-программе	36
2.6. Выполнение SQL-операторов	39
2.7. Выборка результирующих данных	43
2.8. Настройка ODBC	53
3. Доступ к базам данным посредством CGI-скриптов.....	60
3.1. Понятие CGI-скрипта.....	60
3.2. Понятие HTML-формы.....	64
3.3. Переменные CGI-окружения.....	70
3.4. Обработка формы с помощью CGI-скриптов	73
3.5. Доступ к данным из CGI-скрипта, написанного на языке ESQL/C	78
4. Использование языка PHP для доступа к базам данных.....	85
4.1. Синтаксис языка PHP	85
4.2. Работа с формами	92

4.3. Интеграция с базами данных.....	95
5. Использование компонент ADO C++ Builder для доступа к базам данных	107
5.1. Проектирование приложений в интегрированной среде C++Builder.....	107
5.2. Инструменты C++Builder	108
5.3. Работа с проектом	112
5.4. Форма, ее основные свойства, методы, события.....	116
5.5. Обзор популярных компонентов	119
5.6. Работа с базами данных в C++Builder	122
5.7. Обзор ADO-компонентов для работы с БД.....	123
5.8. Компоненты визуализации данных	130
5.9. Обработка ошибочных ситуаций	136
6. Процедурная поддержка ограничений целостности.....	138
6.1. Синтаксис триггеров и триггерных функций в PostgreSQL	138
6.2. Правила использования триггеров при организации процедурной поддержки ограничений целостности	141
7. Анализ данных с использованием сводных таблиц Microsoft Excel	153
7.1. Подготовка результатов базового запроса для последующего анализа	155
7.2. Проведение анализа данных.....	161
Библиографический список.....	169
Приложение 1. Область связи SQL.....	170
Приложение 2. Коды возврата SQLSTATE	171
Приложение 3. Коды возврата протокола HTML.....	176

ВВЕДЕНИЕ

В учебном пособии рассматриваются различные технологии доступа к базам данных современных СУБД, поддерживающих язык SQL. Язык SQL является стандартом в области работы с базами данных, а международные стандарты этого языка позволяют в значительной степени унифицировать средства доступа к данным вне зависимости от используемой СУБД.

Можно выделить несколько уровней доступа к базам данных.

Первый, самый низший, уровень доступа не предполагает каких-либо специальных средств для организации данных, и все пишется на обыкновенном языке программирования (например, Си или Паскале), а данные хранят в обычных файлах, доступ к ним осуществляется посредством системных вызовов `open()`, `close()`, `read()`, `write()`. Хотя такой подход может показаться несколько устаревшим, некоторые специфические задачи обработки достаточно сложной информации можно и нужно решать именно на этом уровне, не привлекая никаких СУБД. Мы не будем далее останавливаться на этом уровне доступа, но принципиальную возможность его существования необходимо иметь в виду.

Следующий, более совершенный, уровень доступа к СУБД предполагает использование интерфейса CLI (Call-Level Interface). Все программирование выполняется на обычном языке третьего поколения, а для работы с данными применяется специально для этого написанная библиотека функций, специфичная для каждой конкретной СУБД. Эти библиотечные функции могут реализовать достаточно сложный метод доступа и поиска данных, индексацию, защиту, разделение данных в многопользовательском режиме. Интерфейс программиста с базой данных состоит из последовательности вызовов функций, в которые нужно вставить требуемый набор параметров. Выглядят такие вызовы примерно так:

```
call open_cursor_for_read (curs,Ncol,buf1,buf2,pole3,pole4,1).
```

В целом интерфейс CLI подобен использованию других стандартных библиотек, к которым программисты уже привыкли.

Корпорация Microsoft в этом направлении пошла дальше, предложив унифицированный CLI-интерфейс, получивший название ODBC (Open Database Connectivity) и позволяющий обращаться к данным любых СУБД и других источников данных. Для общения с базой данных используется язык запросов (SQL). Его синтаксис более понятен программисту, чем выписывание множества вызовов функций со сложными названиями и множеством параметров.

Через ODBC работает и множество программных средств, позволяющих создавать программу, не занимаясь непосредственно кодированием, а используя диалоговый режим, систему меню, мышшь и различные клавиши для получения на выходе окончательного варианта программы. Перечень таких средств обширен: Delphi, C++Builder, Visual C++, Visual Basic и пр.

Следующий уровень доступа, носящий название встроенного SQL (Embedded SQL), предполагает, что программирование ведется также на обычном языке программирования третьего поколения, а операторы языка запросов SQL непосредственно вставляются в текст программы. Естественно, что при этом возникает необходимость использовать препроцессор, который будет читать текст программы со вставленными в нее предписаниями языка запросов и подставлять вместо них вызовы функций СУБД. Полученная после этого программа передается обычному компилятору, который присоединяет к ней библиотеку подпрограмм, реализующих требуемый метод доступа. Примерами средств разработки такого уровня являются INFORMIX-ESQL, UNIFY.

Все современные СУБД работают по технологии «Клиент-сервер». SQL-запрос на получение данных из базы данных может быть послан как из клиентского приложения, так и из предварительно созданной пользователем и размещенной на сервере хранимой процедуры или триггера. В соответствии с этим различают «клиентское» и «серверное» программирование баз данных.

Разрабатываемые клиентские приложения, в свою очередь, можно разбить на две группы: это приложения, запускаемые из среды Internet (так называемые «Web-приложения»), и приложения, запускаемые на компьютере пользователя (в том числе загружаемые по сети), называемые по терминологии IT «тяжелым» клиентом.

Существует целое обилие технологий, языков и средств как для разработки Web-приложений (CGI-, Java-скрипты, языки Perl, PHP и

пр.), так и разработки «тяжелых» клиентов (ADO – Active Data Objects, OLE DB и пр.).

Инструментами «серверного» программирования являются процедуры и триггеры, существенно повышающие эффективность работы с базами данных.

С точки зрения использования сервера все приложения баз данных классифицируются на OLTP-приложения, в задачи которых входят сбор и оперативная обработка данных (задачи учета и текущей обработки данных в различных областях), и OLAP-приложения, направленные на анализ данных и принятие решений.

1. ВСТРОЕННЫЙ SQL (EMBEDDED ESQL/C)

1.1. ОБЩИЕ ПРИНЦИПЫ РАБОТЫ С SQL

ESQL/C – инструмент разработки приложений для пользователей, желающих создавать программный код на языке C с возможностью пользоваться языком SQL. В процессе разработки пишется C-программа, в нее включаются заголовочные файлы, SQL-описания, запускается препроцессор, который преобразует SQL-описания в C-код. Полученный C-код компилируется и линкуется.

Общими для всех языков, использующих встроенный SQL (ESQL/C, 4GL, SPL), являются следующие факторы:

- операторы SQL можно встраивать в исходную программу так, как если бы они были выполняемыми операторами основного языка;
- программные переменные можно использовать в выражениях SQL таким же образом, каким используются литерные значения.

В ESQL/C операторы SQL записываются с помощью предшествующего ему символа \$ либо слова **exec SQL**.

Выполнение оператора SQL фактически является вызовом сервера баз данных. Информация должна передаваться от программы-клиента к программе-серверу и возвращаться обратно. Часть этих взаимодействий осуществляется через так называемые главные переменные (Host-переменные, собственные переменные). В объявлении главных переменных в ESQL/C им предшествует знак \$ или описание производится внутри блока:

<Exec SQL begin declare section . . . Exec SQL end declare section>

Тип и класс хранения главных переменных определяются аналогично C (automatic, static, external). Как и в языке C, возможны массивы (одно-, двумерные), структуры и указатели главных переменных. Инициализация переменных также аналогична C.


```

$int onum;
$double hostdb1;
$char hostarr[80];
$char *hostvar;
$long customer[10];
$struct cust
{
    int c_no;
    char fname[32];
    char lname[32];
} cust_rec;
$struct cust cust2_rec;

```

При использовании главных переменных в SQL-операторах им предшествуют знак \$ или : (стандарт ANSI). Вне описаний главные переменные используются обычным для языка C способом без какого-либо префикса.

```

onum=243;
$delete from items
    where order_num=$onum;
for (i=1; i<10; i++)
    {
        $fetch customer_cursor into :customer[i];
        printf("%s ",customer[i]);
    }
$insert into customer values ($cust_rec);
$insert into customer values ($cust_rec.c_no, $cust_rec.fname
$cust_rec.lname);
$insert into customer values (342, "Smith",$hostvar);

```

Ниже приведена таблица соответствия между ESQL/C-переменными и C-переменными:

SQL-тип	SQL/C-тип	C-тип
Char(n)	—	Char array[n+1]
Date	—	Long int
Datetime	Datetime или dtime t	—
Decimal	Decimal или dec t	—
Smallint	—	Short int
Float	—	Double
Integer, int	—	Long int
Interval	Interval или intrvl t	—
Serial	—	Long int
Small float, real	—	Float

Собственные переменные могут быть использованы как параметры функции посредством указания ключевого слова **parameter**.

```
F(s, id, s_size)      /* s – несобственная переменная */
$parameter char s[20];
    $parameter int id;
    int s_size;
    int s_size;
    $select fname into $s from customer
        where customer_num=$id;
```

Исходный файл с программой на ESQL/C должен иметь расширение **.ec** (например, source.ec).

Вызов транслятора со встроенного ESQL/C выполняется скриптом pgccsi:

```
#!/bin/bash
CC=/usr/bin/gcc
PGPATH=/usr/pgsql-9.3
ECPG=${PGPATH}/bin/ecpg
LFLAGS="-L${PGPATH}/lib"
CFLAGS="-I${PGPATH}/include"
LIBS="-lecp -lecp_compat"
${ECPG} "${1}.ec"
${CC} "${1}.c" -o "${1}.exe" ${LIBS} ${CFLAGS} ${LFLAGS} "${@:2}"
```

Ниже приведен упрощенный вариант синтаксиса команды pgsql:

pgsql <имя файла без расширения>

Сервер баз данных возвращает код результата и, возможно, другую информацию в структуру данных, называемую областью связи SQL (SQL Communication Area – SQLCA). Структура SQLCA описана в заголовочном файле sqlca.h, который автоматически подключается к программе на встроенном SQL/C. Среди других заголовочных файлов можно отметить:

- datetime.h – описывает структуру для типа данных datetime;
- decimal.h – описывает структуру для типа данных decimal;
- locator.h – описывает структуру для blobs-данных;
- varchar.h – описывает структуру для типа данных varchar;
- sqlhdr.h – описывает прототипы функций библиотеки SQL/C;
- sqltype.h, sqltypes.h – структуры для работы с динамическими главными переменными.

1.2. ОБРАБОТКА ОШИБОК

Сервер базы данных всегда возвращает код ошибки и некоторую другую информацию о выполнении операции в структуре данных, называемой областью связи SQL (**SQL Communication Area – sqlca**) (приложение 1).

Поле **sqlerrm** структуры **sqlca** содержит строку **sqlerrmc** с текстовым описанием ошибки и длину строки **sqlerrml** в символах (но не более чем SQLERRMC_LEN).

Поле **sqlstate** содержит символьный код возврата **SQLSTATE** (современная схема кодирования). Пять символов поля **sqlstate** могут содержать цифры и прописные литеры, обозначающие коды ошибок или предупреждений. Первые два символа обозначают класс состояния, последующие три символа уточняют само состояние (приложение 2).

Поле **sqlcode** содержит числовой код возврата **SQLCODE** (предыдущая схема кодирования):

- sqlca.sqlcode = 0 – успешное выполнение (константа ECPG_NO_ERROR);
- 0 < sqlca.sqlcode < 100 – успешное выполнение с дополнительной информацией (в зависимости от описания);

sqlca.sqlcode = 100 – успешное выполнение и наступление события «больше не найдено строк» (константа ECPG_NOT_FOUND);

sqlca.sqlcode < 0 – выполнение с ошибкой (детализированное значение кода ошибки может содержаться в других полях структуры **sqlca**).

Ниже приведен пример функции обработки ошибок. Используемая в примере функция **rgetmsg()** переводит числовой код в текстовое сообщение.

```
void do_error(st_name, errnum)
char *st_name;
int errnum;
{
    char errmsg[400];
    printf("Ошибка %d в %s \n", errnum, st_name);
    rgetmsg(errnum, errmsg, sizeof(errmsg));
    printf("%s\n", errmsg);
    if(sqlca.sqlerrd[1] != 0)
    {
        printf("ISAM-код: %d\n", sqlca.sqlerrd[1]);
        rgetmsg(sqlca.sqlerrd[1], errmsg, sizeof(errmsg));
        printf("Сообщение ISAM: %s\n", errmsg);
    }
    exit(1);
}

.....
$create database personnel;
if(sqlca.sqlcode < 0)
    do_error("Create database", sqlca.sqlcode);
```

1.3. ВЫБОРКА ЕДИНСТВЕННОЙ СТРОКИ

Процедуру выборки единственной строки легко можно пояснить следующим примером:

```
$int avg_price;
...
$select avg(total_price)
into $avg_price from items
where order_num in
(select order_num from orders
where order_date < date("30.06.99"));
printf("Средняя цена %d, avg_price);
...
$char cfname[20], cname[20], ccompany[20];
$int cnumber;
cnumber=104;
$select fname, lname, company
into $cfname, $cname, $ccompany
from customer
where customer_num=$cnumber;
```

Единственное отличие записанных конструкций от классического SQL – наличие спецификатора into. Этот спецификатор задает главные переменные, в которые помещаются данные. Главные переменные используются как получатели данных, так и в спецификаторе Where.

Замечания

1. Если подобный запрос генерирует более одной строки данных, сервер возвращает код ошибки.

2. В запросе SQL не требуется, чтобы главная переменная, в которую помещается значение, имела точно такой же тип данных. Тип

каждой главной переменной передается серверу базы данных, и сервер делает все возможное для преобразования данных из столбцов в форму, используемую переменными-получателями. Однако во время преобразования возможна потеря точности.

1.4. ОБРАБОТКА NULL-ЗНАЧЕНИЙ

Поскольку в языке C нет возможности убедиться, имеет ли элемент таблицы какое-либо значение, ESQL/C делает это для своих главных переменных, называемых переменными-индикаторами. Переменная-индикатор – это дополнительная переменная, ассоциированная с главной переменной, в которую могут поступать NULL-значения. Когда сервер БД помещает данные в главную переменную, он также устанавливает специальное значение в переменную-индикатор, которое показывает, не являются ли эти данные NULL-значением.

Переменная-индикатор описывается как обычная главная переменная целого типа, а при использовании отделяется от главной переменной, в которую передаются значения, знаками «:», «\$» или словом **indicator**.

Примеры равнозначных использований переменных-индикаторов:

```
$int var1:varind1;  
$int var2$varind2;  
$int var3 INDICATOR varind3;
```

При выборе оператором Select NULL-значения переменная-индикатор (если она используется) получает значение -1. В случае нормального возврата переменной индикатор равен 0. Если индикатор не используется, то результат зависит от режима генерации программы:

- если программа компилировалась с флагом -icheck, ESQL/C генерирует ошибку и устанавливает sqlca.sqlcode в отрицательное значение при возврате NULL-значения;
- если программа компилировалась без указанного флага, при отсутствии индикатора ошибка не генерируется.





Ниже приведен фрагмент программы, использующей переменные-индикаторы для обработки NULL-значений.

```
$long op_date:op_date_ind;  
$int the_order;  
.....  
$select paid_date into $opdate : op_date_ind  
      from orders where order_num=$the_order;  
      if (op_date_ind <0)      /* data was null */  
          rstrdate ("01.01.1999", &op_date);
```

Замечание

В данном примере функция **rstrdate()** присваивает главной переменной значения по умолчанию, конвертируя строку в дату.

Ниже приведен набор функций работы с датами:

- 1) int rdatestr(long jdate, char *str); – преобразует дату из внутреннего представления (long) в строку;

- 2) int rstrdate(char *str, long *jdate); – преобразует дату во внутреннее представление (long) из строки;

- 3) int rdayofweek(long jdate): – возвращает день недели из внутреннего представления даты (long);
- 4) int rjulmdy(long jdate, short mdy[3]); – создает массив из трех целых, содержащих месяц, день и год из внутреннего представления даты (long);

- 5) int rmdrjul(short mdy[3], long *jdate); – создает внутреннее представление даты (long) из массива из трех целых чисел (месяц, день и год);

- 6) void rtoday(long *today); – помещает текущую дату в ее внутреннее представление.

1.5. ОБРАБОТКА НЕСКОЛЬКИХ СТРОК

Обработка многострочного запроса осуществляется в два этапа:

- программа запускает запрос, не возвращая никаких данных;
- программа запрашивает строки данных по одной на каждое требование.

Указанные операции выполняются с помощью специального объекта данных, называемого *курсором*, который является структурой данных, отслеживающей внутреннее состояние запроса. Рассмотрим последовательность программных операций при работе с курсором.

1. Программа объявляет курсор и ассоциированный с ним оператор Select. Операция приводит к выделению памяти для хранения курсора (оператор Declare).

2. Программа открывает курсор. Это приводит к началу выполнения ассоциированного оператора Select, а также распознаванию наличия ошибок (оператор Open).

3. Программа считывает строку данных в главные переменные и обрабатывает их (оператор Fetch).

4. Программа закрывает курсор после прочтения последней строки (оператор Close).

Множество строк, возвращаемое предложением Select, носит название *активного множества*.

Курсор может находиться в одном из двух состояний: открытом и закрытом. Когда курсор открыт, он связан с активным множеством и может либо указывать на текущую строку, либо находиться между строк, перед первой строкой, после последней строки. Если курсор находится в закрытом состоянии, то он не связан с активным множеством, хотя и остается связанным с предложением Select.

Объявление курсора. Курсор объявляется с помощью оператора Declare, который задает имя курсора, специфицирует его исполнение и ассоциирует курсор с оператором Select.

```
$nt o_num, i_num, s_num;  
$declare the_item cursor for  
    select order_num, item_num, stock_num  
        into $o_num, $i_num, $s_num from items;
```


В приведенном фрагменте оператор Declare связан с оператором Select, записанным строкой ниже. Оператор Declare устанавливает свойства курсора, выделяет под него память для последующего его использования.

Курсор может задаваться в двух режимах: последовательном и скроллирующем. Последовательный курсор позволяет просматривать активное множество только в последовательном порядке, а также при необходимости удаления или модификации активного множества. Скроллирующий курсор позволяет просматривать строки из активного множества в произвольном порядке.

Пример объявления скроллирующего курсора:

```
Exec sql declare s_curs scroll cursor for
      Select order_num, order_date from orders
      where customer_num > 104
```

Открытие курсора. По оператору Open активизируется курсор, ассоциированный с ним оператор Select передается серверу, который начинает поиск строк до момента конструирования первой строки результата (упреждающий поиск). Оператор не возвращает строк и устанавливает код возврата в переменной sqlcode.

```
$open the_item;
```

Код возврата `sqlcode = 0` означает, что оператор Select синтаксически правилен и курсор готов для использования.

Курсорная выборка строк. Оператор Fetch именуется курсор, подводит его к определенной строке активного множества и выбирает значения из этой строки, а также может задавать имена главных переменных. Способ перехода от строки к строке определяется дополнительными спецификациями.

Имена главных переменных могут быть указаны в спецификации into либо в операторе Select, либо в операторе Fetch (но не в обоих сразу). Вторая форма имеет то преимущество, что разные строки могут быть считаны в разные переменные.

Последовательный курсор выбирает следующую строку из активного множества. Он может только один раз прочитать таблицу, будучи

открытым один раз. Единственной дополнительной спецификацией, применимой к последовательному курсору, является значение по умолчанию next.

Пример использования последовательного курсора:

```
$int o_num, i_num, s_num;
$Declare the_item cursor for select order_num, item_num, stock_num
    into $o_num, $i_num, $s_num from items;
$open the_item;
while (sqlca.sqlcode == 0)
{
    $fetch the_item
    if (sqlca.sqlcode == 0)
        printf("%d %d %d", o_num, i_num, s_num);
}
```

При формировании активного множества для последовательного курсора сервер, стараясь использовать как можно меньше ресурсов, сохраняет только одну строку, которая считается следующей.

Скроллирующий курсор выбирает строку активного множества, используя спецификации:

- next – выбирает следующую строку активного множества;
- previous (prior) – выбирает предыдущую строку активного множества;
- first – выбирает первую строку активного множества;
- last – выбирает последнюю строку активного множества;
- current – выбирает текущую строку активного множества (ранее прочитанную);
- relative – выбирает n -ю строку относительно текущей позиции курсора из активного множества ($n \geq 0$);
- absolute – выбирает n -ю строку активного множества (нумерация с 1).

Примеры записи скроллирующих курсоров:

```
$fetch previous q_curs into $o_num, $i_num, $s_num;
$fetch relative-10 q_curs into $orders;
scan("%d", &row_num;
$fetch absolute $row_num q_curs into $o_num, $i_num, $s_num;
```

Поскольку все строки активного множества прокручиваемого курсора должны сохраняться до закрытия курсора и сервер БД не знает, какая строка будет запрошена в следующий раз, сервер БД реализует активное множество скроллирующего курсора посредством временных таблиц.

Пример использования скроллирующего курсора (выбор каждой второй строки из некоторого набора):

```
$struct customer
{
    char cust_name[20],
    char cust_state[2],
    . . . . .} cust_list[100];
$char wanted_state[2];
int row_count;
$declare cust scroll cursor for
    select * from customer where state=$wanted_state;
printf("Enter 2-letter state code:");
scanf("%s", &wanted_state);
$open cust; row_count=0;
while (sqlca.sqlcode == 0)
    { row_count=row_count+2;
      $fetch relative+2 cust into $cust_list[row_count];    }
$close cust;
```

Объявление массива структур

1.6. ДИНАМИЧЕСКИЙ SQL

С помощью динамического SQL в три этапа формируется оператор SQL с целью его последующего выполнения.

1. Программа собирает текст оператора SQL в виде символьной строки в программной переменной.

2. Она выполняет оператор Prepare, который обращается к серверу баз данных для изучения текста оператора и подготовки его к выполнению.

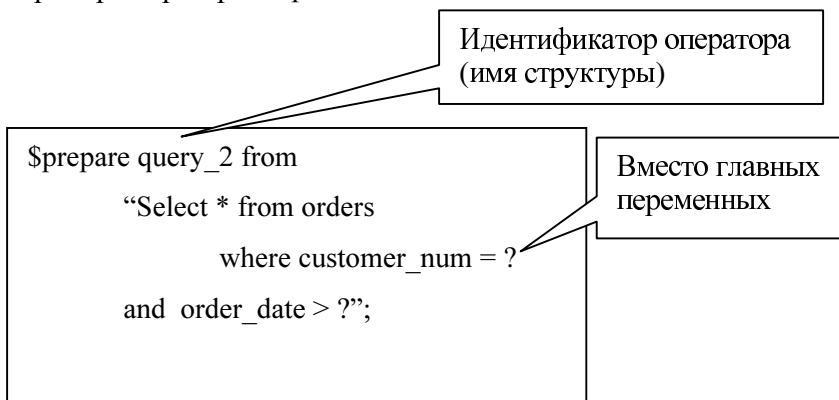
3. Программа использует операторы Execute или Fetch для выполнения подготовленного оператора.

Подготовка оператора. Динамический оператор SQL похож на любой другой оператор SQL, за исключением того, что он не может содержать имена главных переменных. Это приводит к двум ограничениям:

- если динамический оператор является оператором Select, в него нельзя включать спецификатор into (во фразе into должны быть главные переменные, а они не допускаются);
- в любом месте, где по смыслу должна стоять главная переменная, в качестве заполнителя ставят знак вопроса.

В операторе Prepare указываются идентификатор динамически создаваемого оператора и заданный в символьном виде оператор языка SQL, в котором главные переменные заменены знаком вопроса.

Пример оператора Prepare:



Замечание. Запрещенными для динамического формирования являются операторы, непосредственно связанные с динамическим SQL (Prepare, Execute), и операторы управления курсором (Open, Fetch и др.), а также операторы работы с базой данных (Create database, Database и др.).

Результатом подготовки оператора является структура данных, используемая в дальнейшем при выполнении.

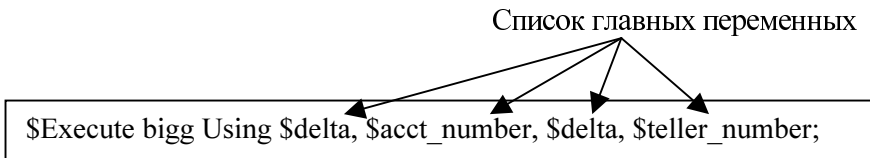
Оператор Prepare не ограничивает символьную строку одним оператором.

```

$char bigquery[100]= "Begin work;";
stcat ("Update account set balance=balance+ ? where ", bigquery);
stcat ("acct_number = ?;", bigquery);
stcat ("Update teller set balance=balance+ ? where ", bigquery);
stcat ("teller_number = ?;", bigquery);
stcat ("Commit work; ", bigquery);
$Prepare bigg from $bigquery;

```

Выполнение подготовленного оператора SQL. Подготовленный оператор можно выполнять многократно. Для выполнения операторов, отличных от Select, используется оператор Execute. В операторе Execute указываются идентификатор оператора и список главных переменных, подставляемых в оператор вместо знака вопроса.



Динамически подготовленный оператор Select подключается к курсору и в дальнейшем используется с помощью курсорных средств. Ниже приведен пример последовательности операторов, динамически подготавливающих оператор SQL и выполняющих его.

```

$int q_c_number, f_o_num;
$long q_o_date, f_o_date;
$char select_2="Select order_num, order_date from orders ";
stcat ("where customer_num = ? and order_date > ?", select_2);
$Prepare q_orders from $select_2;
$Declare cu_orders cursor for q_orders;
$Open cu_orders using $q_c_number, $q_o_date;
$Fetch cu_orders into $f_o_num, $f_o_date;

```

Вместо знаков вопроса

Получатели данных

Последовательность выполнения приведенных выше операторов такова.

1. Символьная строка, содержащая оператор Select, помещается в программную переменную `select_2`. Она содержит два заполнителя, отмеченных знаком «?».

2. Оператор Prepare преобразует символьную строку в структуру данных, связанную с именем `q_orders`.

3. Объявляется курсор `cu_orders` и связывается с именем подготовленного оператора.

4. При открытии курсора начинается выполнение подготовленного оператора. Спецификация `using` в операторе Open представляет имена двух главных переменных, содержимое которых заменяет знаки вопроса в выполняемом операторе.

5. Спецификатор `into` оператора Fetch специфицирует главные переменные, которые должны принимать значения столбцов строки, выбранной по курсору.

Замечание. Указатели позиции «?» нельзя использовать вместо идентификаторов SQL, таких как имя БД, таблицы или столбца: эти идентификаторы должны указываться в тексте оператора при его подготовке. Если эти имена неизвестны при написании оператора, они могут быть получены через пользовательский ввод.

Последнее замечание поясняется следующим примером:

```
$char column_value[40], del_str[100]= "Delete from customer where ";
char column_name[30];
scan ("Enter column name %s", column_name);
stcat ( column_name, del_str);
stcat ("= ?", del_str);
$Prepare del from $del_str;
scan ("Enter a search value in column "+column_name+": %s",
      column_value);
$Execute del using $column_value;
```

Ввод имени столбца

Ввод значения столбца

1.7. ПРИМЕРЫ ИСПОЛЬЗОВАНИЯ ВСТРОЕННОГО SQL

Пример использования журнализованных транзакций при выполнении операций:

```
$short onum;
.....
$Begin work                      /* начать транзакцию */
$Delete from items
    where order_num=$onum;
del_result=sqlca.sqlcode;        /* сохранить два кода */
del_isamno=sqlca.sqlerrd[1];     /* ... ошибки */
del_rowcnt=sqlca.sqlerrd[2];     /* ... и счетчик строк */
if (del_result < 0)              /* в случае ошибки */
    $Rollback work;             /* ... вернуть все назад*/
else                             /* если все о'key */
    $Commit work;               /* ... закончить транзакцию */
```

Пример обновления с использованием курсора: замена имен в названиях некоторой компании:

```
$ char fname[30], lname[30], $company;
$Declare names cursor for
    Select fname, lname, lcompany from customer for update;
$Fetch names into $fname, $lname, $company;
if company == "Sony"
    $Update customer
        set fname="Midory", lname="Tokugawa"
        where current of names;
```

Имена столбцов

Трактуется как обычное условие

Пример удаления строк, содержащих дубликаты записей:

```
Int DelDupOrder()
```

```
{
```

```
    $int ord_num, dup_cnt;
```

```
    int ret_code;
```

```
    $Declare scan_ord cursor for
```

```
        Select order_num, order_date
```

```
        into $ord_num
```

```
        from orders for update;
```

```
    $Open scan_ord;
```

```
    if (sqlca.sqlcode != 0) return (sqlca.sqlcode);
```

```
    $Begin work;
```

```
    for(;;)
```

```
    { $Fetch next scan_ord;
```

```
      if (sqlca.sqlcode != 0) break;
```

```
      dup_cnt = 0; /* значение по умолчанию в случае ошибки */
```

```
      $Select count(*) into $dup_cnt from orders
```

```
        where order_num=$ord_num;
```

```
      if (dup_cnt > 1 )
```

```
      { $Delete from orders
```

```
        where current of scan_ord;
```

```
        if (sqlca.sqlcode != 0) break; }
```

```
    }
```

```
        ret_code=sqlca.sqlcode;
```

```
    if (ret_code == 100) /* просто конец данных */
```

```
        $Commit work;
```

```
    else /* ошибка при выборке или при удалении */
```

```
        $Rollback work;
```

```
    return (ret_code);
```

```
}
```

Объявляется курсор scan_ord для просмотра строк таблицы Orders

Спецификация for update означает, что курсор может быть использован для модификации данных. Номер заказа запоминается в ord_num

Если курсор объявлен правильно, функция начинает транзакцию и проходит в цикле по строкам

Для каждой строки используется оператор Select, определяющий, сколько строк таблицы содержат номер заказа такой же, как и текущая строка. Если есть дубли, они удаляются

2. ДОСТУП К ДАННЫМ НА ОСНОВЕ СТАНДАРТА ODBC

Open Database Connectivity (ODBC) – стандарт прикладного программного интерфейса API (Application Programming Interface), который позволяет программам, работающим в различных средах, взаимодействовать с произвольными СУБД как персональными, так и многопользовательскими, функционирующими в различных операционных системах. Основная цель ODBC – сделать взаимодействие приложения и СУБД прозрачным, не зависящим от класса и особенностей используемой СУБД.

2.1. СТАНДАРТИЗАЦИЯ АРХИТЕКТУРЫ ДОСТУПА К БАЗЕ ДАННЫХ

Работа с СУБД в локальной сети требует ряда компонентов, каждый из которых независим от вышележащего слоя и поддерживает определенный интерфейс программирования (рис. 2.1).

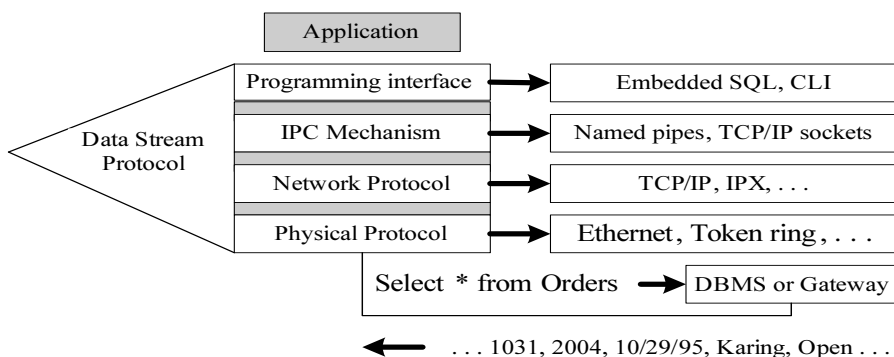


Рис. 2.1. Сетевой аспект архитектуры доступа к базе данных

Интерфейс программирования содержит обращения, сделанные прикладной программой посредством вложенного SQL или интерфейса CLI.

Протокол передачи данных – это логический протокол, описывающий поток данных, перемещаемых между СУБД и пользователем и не зависящий от протоколов, используемых сетью. Например, протокол мог бы требовать, чтобы первый байт описывал то, что содержится в остальной части потока: выполняемый оператор SQL, возвращаемое значение ошибки или данные. Формат остаточной части данных в потоке зависел бы от этого флажка. Ошибка, в свою очередь, могла бы кодироваться целочисленным кодом ошибки и битами, определяющими целочисленную длину сообщения в ошибке, и т. д.

Коммуникации между процессами обеспечиваются механизмом IPC (Inter Process Communication). Например, могут использоваться именованные каналы, сокеты TCP/IP или DECnet. Выбор механизма IPC зависит от типа используемой сети и операционной системы.

Транспортный протокол используется для передачи данных по сети. К транспортным протоколам относятся TCP/IP, DECnet, SPX/IPX, и они являются специфическими для каждой сети.

Наконец, физический (канальный) протокол реализуется средствами аппаратного обеспечения, он никак не связан с программным обеспечением.

Если внимательно посмотреть на описанные выше компоненты, то можно отметить, что два из них – программный интерфейс и потоковый протокол – являются хорошими кандидатами на стандартизацию. Три других компонента – механизм IPC, транспортный и физический протоколы – слишком зависят от сети и операционной системы и находятся на недопустимо низком уровне в 7-уровневой сетевой модели. Именно программный интерфейс был выбран в качестве основы для стандартизации.

Поскольку интерфейс CLI может быть реализован через библиотеки или драйверы базы данных, а современные операционные системы могут загружать такие библиотеки и вызывать из них необходимые функции во время выполнения, прикладная программа может обращаться не только к данным из различных СУБД, но и к данным из многих баз данных одновременно.

2.2. АРХИТЕКТУРА И УРОВНИ СООТВЕТСТВИЯ ODBC

Общая схема архитектуры ODBC показана на рис. 2.2. Приложение имеет пять логических слоев: прикладной слой, интерфейс ODBC, диспетчер драйверов, драйвер и источник данных.

Прикладной слой реализует GUI и бизнес-логику. Он написан на языке программирования, таком как C++, Java или Visual Basic. Приложение использует функции из интерфейса ODBC для взаимодействия с базами данных.

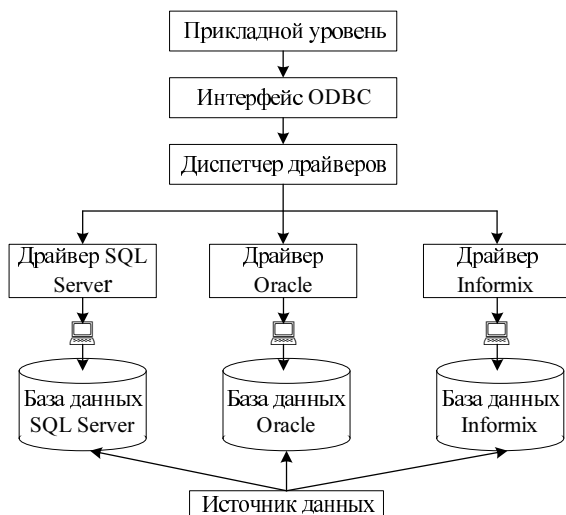


Рис. 2.2. Архитектура ODBC

Диспетчер драйверов является частью ODBC Microsoft. Он управляет различными драйверами, имеющимися в системе, выполняя их загрузку, перенаправляет вызовы на нужный драйвер и предоставляет прикладной программе информацию о драйвере, когда это необходимо. Поскольку одна прикладная программа может быть связана с несколькими базами данных, диспетчер драйверов гарантирует, что соответствующая система управления базой данных получит все запросы, направляемые к ней.

Драйвер – это часть архитектуры, которая все знает о какой-либо базе данных. Обычно драйвер связан с конкретной базой данных, например, Access, Oracle, SQL Server, Informix, DB2, Sybase. Интерфейс ODBC имеет набор функций для выполнения операторов SQL, управления соединением, получения информации о базе данных и т. д.

Источник данных в контексте ODBC может быть системой управления базой данных или просто набором файлов на жестком диске.

Уровни соответствия ODBC приведены в таблице.

Уровни соответствия ODBC

Тип соответствия	Уровень соответствия	Описание
Соответствие API	Уровень ядра	Включает функции: <ul style="list-style-type: none"> • выделения и освобождения описателей связи, SQL-оператора и окружения • получения результата и служебной информации • получения информации об ошибках • способности выполнять транзакции и их откат
	Уровень 1	Включает возможности уровня ядра, дополненные функциями: <ul style="list-style-type: none"> • посылки и получения частичных наборов данных • поиска информации в каталоге • получения информации о возможностях драйвера и базы данных
	Уровень 2	Включает возможности уровня 1, дополненные функциями: <ul style="list-style-type: none"> • обработки массивов как параметров • прокрутки курсора • вызова DDL транзакций
Соответствие грамматике SQL	Минимальная грамматика	Включает функции: <ul style="list-style-type: none"> • создания и удаления таблиц • простые функции выбора, вставки, модификации и удаления • простые выражения
	Грамматика ядра	Включает возможности минимальной грамматики, дополненные функциями: <ul style="list-style-type: none"> • изменения таблиц • создания и удаления индексов • создания и удаления логических таблиц базы данных для DDL • в выражениях, например SUM() и MAX()
	Расширенная грамматика	Включает возможности грамматики ядра, дополненные: <ul style="list-style-type: none"> • функциями создания внешних соединений • возможностями позиционированных модификации и удаления • выражениями и новыми типами данных • возможностью вызова процедур

2.3. ФУНКЦИИ ODBC И СТРУКТУРА КОМАНД

Основной алгоритм создания ODBC-программ для наглядности можно представить в виде следующей блок-схемы (рис. 2.3).



Рис. 2.3. Последовательность действий при создании ODBC-программы

ODBC имеет богатый выбор функций: от простых операторов соединения до процедур, выдающих целый набор результатов. На рис. 2.4 приведена последовательность команд ODBC для связи с базой данных, выполнения оператора SQL, обработки данных, полученных в результате SQL-запроса, и закрытия связи.

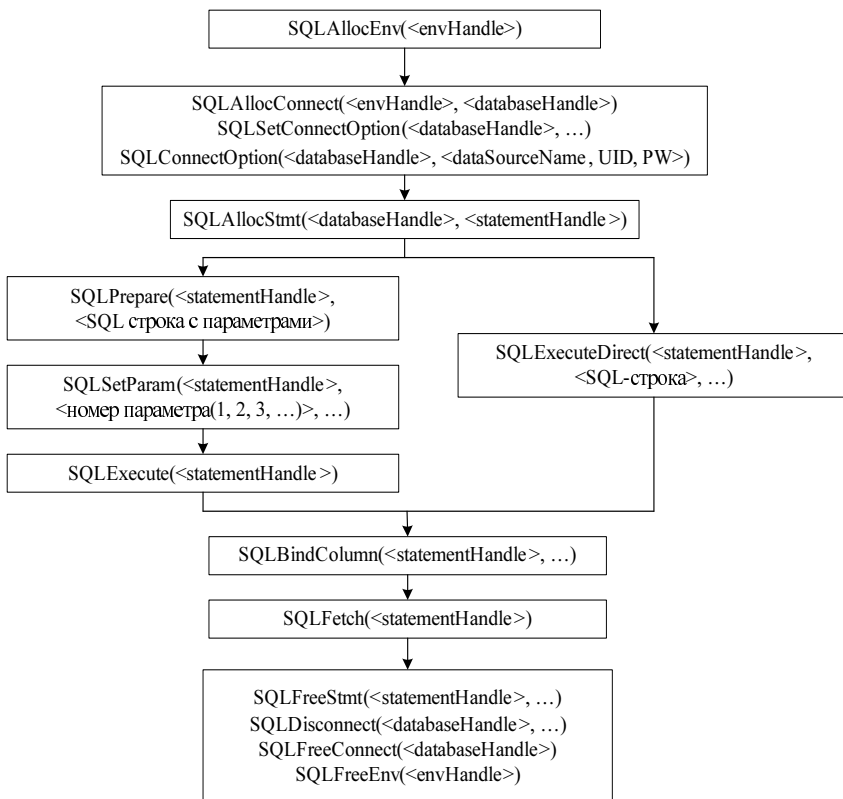


Рис. 2.4. Базовый набор ODBC-функций

Все функции ODBC имеют префикс SQL и один или несколько параметров, которые могут быть входными (информация для драйвера) или выходными (информация от драйвера).

2.4. ПОДГОТОВИТЕЛЬНЫЕ ОПЕРАЦИИ В ODBC-ПРОГРАММЕ

Назначение идентификатора окружения

Прежде чем прикладная программа сможет вызвать какую-либо функцию ODBC, необходимо выполнить инициализацию ODBC и установить окружение. Поскольку в рамках одного окружения можно

задать произвольное число соединений, для каждой прикладной программы достаточно установить одно окружение. Функция ODBC **SQLAllocEnv()**, выполняющая такое назначение окружения, распределяет память для идентификатора окружения и инициализирует интерфейс ODBC на уровне вызовов для использования прикладной программой. Эту функцию необходимо вызывать, прежде чем любую другую из функций ODBC.

RETCODE	SQLAllocEnv (env);
HENV env;	– указатель области хранения в памяти идентификатора окружения

Освобождение идентификатора окружения

Когда прикладная программа заканчивает использование ODBC, необходимо, чтобы было освобождено связанное с ней окружение. Функция **SQLFreeEnv()**, параметром которой является идентификатор окружения, освобождает ресурсы, зарезервированные для данной прикладной программы:

RETCODE	SQLFreeEnv (env);
HENV env;	– имя идентификатора окружения, который освобождается

Назначение идентификатора соединения

Идентификатор соединения представляет собой соединение между прикладной программой и источником данных, с которым прикладная программа предполагает соединиться. Если требуется установить соединение с двумя источниками данных, то необходимо назначить два идентификатора соединения. Функция, распределяющая память для заданного идентификатора окружения, носит имя **SQLAllocConnect()**:

RETCODE	SQLAllocConnect (env, dbc);
HENV env;	– указатель на идентификатор окружения прикладной программы;
HDBC dbc;	– указатель области хранения памяти для идентификатора соединения

Соединение с источником данных

Несмотря на то что существует много функций ODBC для установки соединения, на базовом уровне API такая функция одна – это **SQLConnect()**, которая обеспечивает простой и эффективный способ соединения с источником данных. Поскольку все драйверы поддерживают **SQLConnect()**, доступ с помощью этой функции осуществляется наилучшим образом. **SQLConnect()** загружает драйвер базы данных и устанавливает соединение с источником данных. Идентификатор соединения ссылается на местоположение области хранения всей информации о соединении, включая ее статус, состояние транзакции и информацию об ошибке.

```
RETCODE SQLConnect (dbc, szDSN, sbDSN, szUID, sbUID,
                    szAuthStr, cbAuthStr);
HDBC dbc;          – идентификатор соединения;
UCHAR szDSN;       – строка с именем источника данных, с которым
                    прикладная программа собирается соединиться;
SWORD sbDSN;       – длина строки источника данных либо значение
                    SQL_NTS;
UCHAR szUID;        – имя пользователя;
SWORD sbUID;        – длина имени пользователя либо значение
                    SQL_NTS;
UCHAR szAuthStr;    – пароль пользователя;
SWORD cbAuthStr;    – длина пароля или SQL_NTS
```

Замечание. Значение SQL_NTS является константой ODBC, которая используется вместо длины параметра в случае, если параметр содержит строку с нулевым окончанием.

Разъединение с источником данных

Как только прикладная программа заканчивает использование доступа к источнику данных, она должна отсоединиться от него, чтобы дать другим пользователям войти в систему. Этой операцией управляет функция **SQLDisconnect()**, которая закрывает соединение с источником данных с помощью специального идентификатора соединения.

RETCODE	SQLDisconnect (dbc);
HDBC dbc;	– идентификатор доступа для отсоединения

Освобождение идентификатора соединения

Если прикладная программа не планирует относительно долгое время использовать идентификатор соединения, она должна освободить этот идентификатор и назначенные ему ресурсы. Данную операцию выполняет функция **SQLFreeConnect()**.

RETCODE	SQLFreeConnect (dbc);
HDBC dbc;	– указатель области памяти для освобождаемого идентификатора соединения

Назначение идентификатора оператора

Любая функция SQL, которая имеет отношение к обработке или передаче SQL-операторов, требует в качестве параметра идентификатор оператора. Идентификатор оператора аналогичен идентификатору окружения или соединения, за исключением того, что он ссылается на SQL-оператор или другие функции ODBC, которые возвращают результаты. Следует заметить, что идентификатор соединения может быть связан с несколькими идентификаторами операторов, но каждый идентификатор оператора связан только со своим идентификатором соединения. Для того чтобы назначить идентификатор оператора, прикладной программе нужно вызвать функцию **SQLAllocStmt()**, которая выделит память для SQL-операторов; это необходимо сделать до использования операторов.

RETCODE	SQLAllocStmt (dbc, stmt);
HDBC dbc;	– идентификатор соединения;
HSTMT stmt;	– указатель области хранения в памяти для идентификатора оператора

Освобождение идентификатора оператора

Для освобождения идентификатора оператора используется функция **SQLFreeStmt()**, которая выполняет следующие действия:

- останавливает любые SQL-операторы, которые в данный момент обрабатываются и связаны с заданным идентификатором оператора;
- закрывает любые открытые курсоры, которые имеют отношение к определенному идентификатору оператора;
- отбрасывает ожидаемые результаты;
- освобождает все ресурсы, связанные с определенным идентификатором оператора.

RETCODE	SQLFreeStmt (stmt, fOption);
HSTMT stmt;	– идентификатор оператора;
UWORD fOption;	– одна из опций, перечисленных ниже

Опции функции **SQLFreeStmt()** имеют следующие значения.

1. **SQL_CLOSE** – закрывает курсор, связанный с stmt (если он был определен), и отбрасывает все ожидаемые результаты. Прикладная программа может открыть этот курсор позднее и выполнить оператор **Select** с теми же или другими значениями параметров. Если курсор не открыт, то эта опция не влияет на программу.

2. **SQL_DROP** – освобождает stmt и все ресурсы, связанные с ним, закрывает курсор, если он открыт, и отбрасывает все ожидаемые строки. Эта опция завершает все обращения к stmt. Оператор stmt обязательно должен быть переназначен для повторного использования.

3. **SQL_UNBIND** – освобождает все буферы столбцов, которые повторно используются функцией **SQLBindCol()** для данного идентификатора оператора.

4. **SQL_RESET_PARAMS** – освобождает все буферы параметров, которые были установлены функцией **SQLBindCol()** для данного идентификатора оператора.

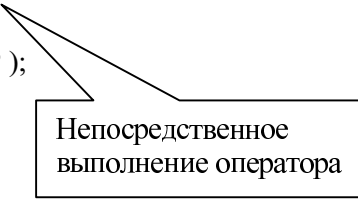
Приведем пример простой ODBC-программы.

```

#include <sqlext.h>
#include <sqltypes.h >
#include <strings.h>
void main()
{
    HENV    env;
    HDBC    dbc;
    HSTMT   stmt;
    UCHAR   driver[32];

    . . . .
    SQLAllocEnv (&env );
    SQLAllocConnect (env, &dbc );
    strcpy((char*)driver,"MYDB");
    SQLConnect (dbc, driver, SQL_NTS, "name", SQL_NTS,
                "password", SQL_NTS);
    SQLAllocStmt (dbc, &stmt );
    SQLExecDirect (stmt, "DELETE FROM S WHERE n_post=5",
                  SQL_NTS);
    SQLFreeStmt ( stmt, SQL_DROP );
    SQLDisconnect ( dbc );
    SQLFreeConnect ( dbc );
    SQLFreeEnv ( env );
}

```



Непосредственное
выполнение оператора

Сделаем некоторые промежуточные выводы: в каждой ODBC-программе необходимо строго придерживаться определенной последовательности описания функций, отвечающих за окружение, соединение и назначение операторов:

- назначение идентификатора окружения;
- назначение идентификатора соединения;

- соединение с сервером;
- назначение идентификатора оператора;
- выполнение оператора и выборка данных;
- освобождение идентификатора оператора;
- разрыв соединения с сервером;
- освобождение идентификатора соединения;
- освобождение идентификатора окружения.

2.5. ОБРАБОТКА ОШИБОК В ODBC-ПРОГРАММЕ

Важной частью любого API является его способность возвращать коды и состояния ошибок. Отлаженная система сообщений об ошибках позволяет прикладному программному обеспечению определить причину ошибки при выполнении какой-либо функции, а затем ее откорректировать.

Коды возврата. Все функции ODBC возвращают коды возврата. В случае ошибки код возврата не сообщает точной причины ее возникновения, а указывает лишь на то, что ошибка имела место вообще. Ниже приведены основные значения кодов возврата RETCODE, которые могут возвращаться ODBC-функциями.

- **SQL_SUCCESS** – ошибка отсутствует.
- **SQL_SUCCESS_WITH_INFO** – функция обработки завершена, но при вызове вспомогательной функции **SQLError()** идентифицируется ошибка. В большинстве случаев это возникает, когда значение, которое должно быть возвращено, большего размера, чем это предусмотрено буфером прикладной программы.
- **SQL_ERROR** – функция не была завершена из-за возникшей ошибки. При вызове функции **SQLError()** можно будет получить больше информации о сложившейся ситуации.
- **SQL_INVALID_HANDLE** – неправильно определен идентификатор окружения, соединения или оператора. Ситуация часто имеет место, когда идентификатор используется после того как он был освобожден, или если был определен нулевой указатель.
- **SQL_NO_DATA_FOUND** – невозможно выбрать данные. Ситуация фактически не является ошибочной и чаще всего возникает при использовании курсора, когда больше нет строк для его продвижения.
- **SQL_NEED_DATA** – необходимы данные для параметра.

Обработка ошибок. В случае когда прикладная программа передает неверные значения в драйвер ODBC, в СУБД возникают проблемы. Хорошо написанная программа может восстанавливаться из ошибочного состояния, если она знает место возникновения ошибки. В ODBC имеются расширенные средства для исправления ошибки с использованием функции **SQLError()**. Следует отметить, что **SQLError()** возвращает дополнительную информацию лишь в том случае, если кодом возврата функции обработки являются **SQL_ERROR** или **SQL_SUCCESS_WITH_INFO**.

RETCODE SQLError (henv, hdbc, hstmt, szSqlState, pfNativeError, szErrorMsg, cbErrorMsgMax, cbErrorMsg);

HENV henv; – идентификатор окружения;

HDBC hdbc; – идентификатор соединения;

HSTMT hstmt; – идентификатор оператора;

UCHAR szSqlState; – **SQLSTATE** в качестве кода завершения (некоторые из кодов приведены в *Приложении 2*);

SDWORD pfNativeError; – переменная, в которой будет возвращена ошибка, возникшая в СУБД, а также ее собственный код (код СУБД);

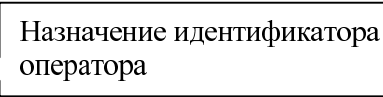
UCHAR szErrorMsg; – указатель на буфер, куда будет возвращен текст ошибки (строка с нулевым окончанием).

SWORD cbErrorMsgMax; – максимальный размер описанного выше буфера, который должен быть меньше переменной или равен **SQL_MAX_MESSAGE_LENGTH-1**;

SWORD cbErrorMsg; – число байт, скопированных в буфер

Фрагмент программы обработки ошибок с использованием функции **SQLError()** приведен ниже.

```
HDBC dbc;
HSTMT stmt;
RETCODE rc;
UCHAR  errmsg[SQL_MAX_MESSAGE_LENGTH];
SDWORD nativeerr;
SWORD  actualmsglen;
rc = SQLAllocStmt ( dbc, &stmt );
if ( rc == SQL_ERROR )
    {
        SQLError (env, dbc, stmt, sqlstate, &nativeerr, errmsg,
                  SQL_MAX_MESSAGE_LENGTH - 1,
                  &actualmsglen);
        printf ("SQLSTATE = %s\n      , sqlstate, "Native error =
        %d\n", nativeerr);
        errmsg[actualmsglen] = '\0';
        printf ("MSG = %s\n\n",errmsg); }
    return; }
rc = SQLExecDirect (stmt, "DELETE FROM S WHERE n_post=5",
SQL_NTS );
if ( rc == SQL_ERROR )
    {
        ....
    return; }
```



Назначение идентификатора оператора

Таким образом, чтобы прикладная программа могла восстановиться из любого ошибочного состояния, важно знать место возникновения ошибки. Для этого используют коды возврата RETCODE и одну из базовых функций API – **SQLError()**, с помощью которой можно получить подробное описание произошедшей ошибки.

2.6. ВЫПОЛНЕНИЕ SQL-ОПЕРАТОРОВ

Существуют два метода, которые может использовать прикладная программа для выполнения SQL-операторов: непосредственное выполнение и подготавливаемое.

Непосредственное выполнение является самым быстрым и простым способом использования SQL-операторов, однако подготавливаемое выполнение обеспечивает большую гибкость, особенно при использовании SQL-операторов с параметрами.

Непосредственное выполнение используется в следующих случаях:

- 1) SQL-операторы выполняются только один раз;
- 2) не требуется информации о результирующем множестве до выполнения оператора.

Если хотя бы одно из условий не соблюдается, подготавливаемое выполнение с помощью функций **SQLPrepare()** и **SQLExecute()** предпочтительно.

Непосредственное выполнение осуществляется с помощью функции **SQLExecDirect()**, среди параметров которой задается SQL-оператор, подготовленный для использования драйвером.

RETCODE SQLExecDirect (hstmt, szSqlStr, cbSqlStr);	
HSTMT hstmt;	– идентификатор оператора;
UCHAR szSqlStr;	– строка с SQL-оператором;
SDWORD cbSqlStr;	– длина строки szSqlStr

В следующем примере демонстрируются различные способы использования функции **SQLExecDirect()**.

UCHAR sql[255];
Rc = SQLExecDirect(hstmt, "Select * from Inventory", SQL_NTS);
.....
strcpy((char*)sql, "delete from s where n_post='S1' ");
rc=SQLExecDirect (stmt, sql, (SDWORD)strlen((char*)sql));

Подготавливаемое выполнение предпочтительно по сравнению с непосредственным, если требуется выполнить SQL-оператор более одного раза или необходима информация о результирующем множестве до выполнения оператора.

Для подготавливаемого выполнения SQL-оператора необходимы две функции: **SQLPrepare()** и **SQLExecute()**. Прикладная программа для подготовки выполнения оператора сначала вызывает функцию **SQLPrepare()** с параметром, являющимся SQL-оператором.

```
RETCODE SQLPrepare (hstmt, szSqlStr, cbSqlStr);  
HSTMT hstmt;           – идентификатор оператора;  
UCHAR szSqlStr;        – строка с SQL-оператором;  
SDWORD cbSqlStr;       – длина строки szSqlStr
```

Затем вызывается функция **SQLExecute()**, чтобы этот оператор выполнить.

```
RETCODE SQLExecute (hstmt);  
HSTMT hstmt;           – идентификатор оператора
```

Использование параметров. Параметры могут использоваться как при непосредственном выполнении (**SQLExecDirect()**), так и при подготавливаемом (**SQLPrepare()**, **SQLExecute()**). Использование параметров в обоих случаях дает дополнительное преимущество, позволяя писать более гибкие прикладные программы. Маркеры параметров определяются в SQL-операторах с помощью знака «?». Например,

```
SELECT n_post FROM s WHERE town=?   или  
INSERT INTO P(name, town) VALUES (?, ?)
```

Для того чтобы связать буфер с маркерами параметров, прикладная программа должна вызвать функцию **SQLBindParameter()**:

```
RETCODE SQLBindParameter (hsmt, ipar, fParamType,  
                           fCType, fSqlType, cbColDef, ibScale,  
                           rgbValue, cbValueMax, pcValue)
```


Ниже перечислены параметры функции **SQLBindParameter()**:

- HSTMT hstmt; – идентификатор оператора, совпадающий со значением идентификатора, с которым этот оператор подготавливается и выполняется.
- UWORD ipar; – номер параметра для связи. В операторе SQL операторы нумеруются слева направо, начиная с единицы. Например, если для SQL-оператора используются три параметрических маркера:
- insert into j (n_izd, name, town) values (?, ?, ?) ,**
- то для связи этих параметров функция **SQLBindParameter()** вызывается три раза с параметром ipar, установленным в 1, 2 и 3 соответственно.
- SWORD fParamType; – тип параметра для связи, принимающий одно из трех значений:
SQL_PARAM_INPUT – применяется для процедур, использующих параметры ввода;
SQL_PARAM_INPUT_OUTPUT – маркирует параметр ввода/вывода в процедуре;
SQL_PARAM_OUTPUT – маркирует значение возврата или параметр вывода в процедуре.
- SWORD fCType; – C-тип данных параметра, из которого необходимо конвертировать данные.
- SWORD fSqlType; – ODBC-тип данных параметра, в который конвертируются данные и который должен совпадать с SQL-типом столбца, соответствующего этому параметрическому маркеру.
- UDWORD cbColDef; – точность столбца или выражения соответствующего маркера параметра.
- SWORD ibScale; – размер столбца или выражения соответствующего маркера параметра.
- PTR rgbValue; – указатель буфера для данных параметра, который при вызове **SQLExecute()** или **SQLExecuteDirect()** содержит действительные значения параметра.

SDWORD cbValueMax; – максимальная длина буфера rgbValue.
SDWORD pcbValue; – указатель буфера для длины параметра.

Использование метода подготавливаемого выполнения с передачей параметров иллюстрируется ниже.

```
HSTMT stmt;  
HDBC dbc;  
RETCODE rc;  
UCHAR param1[10], param2[10];  
SDWORD cbTEST = SQL_NTS;  
  
...  
  
// подготовка SQL-оператора с двумя параметрами  
rc = SQLAllocStmt (dbc, &stmt);  
strcpy ((char*)sql, "INSERT INTO s(name, town) VALUES (?,?)");  
rc = SQLPrepare (stmt, sql, (SDWORD)strlen((char*)sql));  
rc = SQLBindParameter (stmt, 1, SQL_PARAM_INPUT,  
SQL_C_CHAR, SQL_CHAR, 10, 0, param1, 0, &cbTEST);  
rc = SQLBindParameter (stmt, 2, SQL_PARAM_INPUT,  
SQL_C_CHAR, SQL_CHAR, 10, 0, param2, 0, &cbTEST);  
  
// определение значения параметров  
strcpy(param1, "Джон");  
strcpy(param2, "Рим");  
rc = SQLExecute (stmt);
```

Подготовка оператора к выполнению

Связь буфера с маркерами параметров

Выполнение оператора

Ниже приведены соотношения основных значений C- и SQL-типов параметров.

C-тип		SQL-тип	
SQL_C_BINARY	SQL_C_FLOAT	SQL_BINARY	SQL_DOUBLE
SQL_C_BIT	SQL_C_TIME	SQL_BIT	SQL_FLOAT
SQL_C_CHAR	SQL_C_DEFAULT	SQL_CHAR	SQL_INTEGER
SQL_C_DATE	SQL_C_SLONG	SQL_DATE	SQL_REAL
SQL_C_DOUBLE	SQL_C_SSHORT	SQL_DECIMAL	SQL_SMALLINT
		SQL_TIME	SQL_VARCHAR

2.7. ВЫБОРКА РЕЗУЛЬТИРУЮЩИХ ДАННЫХ

При выполнении SQL-оператора формируется результирующее множество (для оператора Select) или число строк, на которые воздействовал оператор (для операторов Update, Insert, Delete). Исходя из этого рассматриваемые в данном разделе ODBC-функции можно разбить на три группы:

- информационные функции результирующих множеств;
- базовые функции выборки данных;
- расширенные функции выборки данных.

Под результирующим множеством понимается набор строк и столбцов, которые были определены SQL-оператором.

После того как был выполнен SQL-оператор и создано результирующее множество, оно тем не менее располагается на сервере и должно быть извлечено и сохранено с помощью средств прикладной программы клиента.

Информационные функции результирующих множеств

Среди ODBC-функций данной группы отметим **SQLRowCount()** и **SQLNumResultCol()**. Первая из них возвращает число строк, на которые воздействовал SQL-оператор (–1 в случае, если число таких строк не определяется), вторая – возвращает число столбцов в результирующем множестве.

```

RETCODE SQLRowCount (hstmt, psrow);
HSTMT stmnt;                – идентификатор оператора;
SDWORD FAR* psrow;          – число строк, на которые
                             воздействовали операторы Update,
                             Insert, Delete;
RETCODE SQLNumResultCount (hstmt, pcol);
HSTMT stmnt;                – идентификатор оператора;
SDWORD FAR* pcol;           – число столбцов в результирующем
                             множестве

```

Замечание. Хотя при использовании функции **SQLRowCount()** некоторые драйверы способны возвращать число строк, которые были возвращены оператором Select, этот способ в общем случае не является корректным, поскольку достаточно много источников данных не могут возвращать число строк до их извлечения.

Ниже демонстрируется использование функции **SQLRowCount()** после удаления строк из таблицы для выяснения того, на сколько строк воздействовал оператор удаления.

Выполнение
оператора Delete

```

...
SQLExecDirect (hstmt, "Delete from Employee Where
Last_name='Signore'");
SQLRowCount (hstmt, &iNumRows);

```

Базовые функции выборки данных

После генерации результирующего множества строки в нем имеют определенный порядок, который используется для продвижения по результирующему множеству. Например, конкретная строка должна быть первой, третьей, четвертой, седьмой и т. д. Чтобы выполнить такое продвижение в пределах результирующего множества, программа должна использовать курсор.

Курсор автоматически генерируется для каждого результирующего множества. Непосредственно после создания результирующего множества курсор устанавливается перед первой строкой данных. Для продвижения курсора используются функции ODBC **SQLFetch()** или **SQLExtendedFetch()**. **SQLFetch()** продвигает курсор вперед на каждую строку до тех пор, пока он не будет установлен за последней строкой результирующего множества. **SQLExtendedFetch()** по умолчанию делает то же самое, однако имеет опцию, которая позволяет продвинуть курсор в произвольном направлении.

В ODBC для выборки результатов существуют две функции базового уровня: **SQLBindCol()** и **SQLFetch()**. Первая определяет область хранения данных результирующего множества, вторая – осуществляет выборку в область хранения, при этом **SQLBindCol()** может быть вызвана в любой момент времени, даже до того, как оператор ожидает результирующее множество.

Каждый столбец, который требуется выбрать, связывается с помощью отдельного вызова функции **SQLBindCol()**. Функция **SQLBindCol()** назначает область хранения в памяти и тип данных для столбца результирующего множества. Она определяет:

- буфер хранения для получения содержимого столбца данных в результирующем множестве;
- длину указанного буфера хранения;
- область памяти для хранения длины столбца выборки;
- преобразование типа данных.

Алгоритм программы, использующей **SQLFetch()** и **SQLBindCol()** для возвращения данных из результирующего множества, предполагает выполнение следующих шагов.

1. Вызывается функция **SQLBindCol()** один раз для каждого столбца, который должен быть возвращен из результирующего множества.

2. Вызывается функция **SQLFetch()** для перемещения курсора на следующую строку и возврата данных из связанных столбцов.

3. Повторяется шаг 2 до тех пор, пока функция **SQLFetch()** не возвратит **SQL_NO_DATA_FOUND**. Это указывает на то, что был достигнут конец результирующего множества. Если результирующее множество является пустым, то **SQL_NO_DATA_FOUND** будет возвращен при первом вызове **SQLFetch()**.

RETCode SQLBindCol	(hstmt, icol, fcType, rgbValue, cbValueMax, pcbValue);
HSTMT stmt;	– идентификатор оператора;
UWORD icol;	– номер столбца результирующего множества, упорядоченного слева направо, начиная с 1;
SWORD fcType;	– C-тип данных результирующего множества;
PTR rgbValue;	– указатель области хранения данных. Если rgbValue является нулевым указателем, то драйвер “отвязывает” столбец. Для “отвязывания” всех столбцов прикладная программа вызывает функцию SQLFreeStmt() с опцией SQL_UNBIND;
SDWORD cbValueMax;	– максимальная длина буфера rgbValue. Для символьных данных rgbValue должен также включать в себя место для байта нулевого окончания;
SDWORD FAR* pcbValue;	– число байт, которое может возвращаться в rgbValue при вызове SQLFetch() .

После того как все требуемые столбцы связаны, посредством функции **SQLFetch()** выполняется выборка данных в определенную область хранения.

RETCode SQLFetch	(hstmt);
HSTMT stmt;	– идентификатор оператора

Использование функций **SQLBindCol()** и **SQLFetch()** иллюстрируется следующим примером:

```
HSTMT stmt;
HDBC dbc;
SDWORD val_name, val_reiting;
UCHAR name[20], town[15];
int reiting;
...
SQLAllocStmt (dbc, &stmt);
SQLExecDirect (dbc, "SELECT name, reiting FROM S", SQL_NTS);
SQLBindCol (stmt, 1, SQL_C_CHAR, &name,
            (SDWORD)sizeof(name), &val_name);
SQLBindCol (stmt, 2, SQL_C_SSHORT, &reiting,
            (SDWORD)sizeof(town), &val_reiting);
for (;;)
{
    rc = SQLFetch (stmt);
    if (rc == SQL_NO_DATA_FOUND)
    { printf ("SQLFetch возвратила: SQL_NO_DATA_FOUND\n");
      break; }
    if ((rc != SQL_SUCCESS) && (rc != SQL_SUCCESS_WITH_INFO))
    { printf ("Ошибка при вызове SQLFetch\n");
      break; }
    printf ("% -20s,% -15s\n ", name, town);
}
SQLFreeStmt (stmt, UNBIND);
```

Назначение идентификатора оператора

Непосредственное выполнение оператора

Определение области хранения результирующих данных

Выборка строки данных

Освобождение идентификатора оператора

Дополнительно к функциям **SQLBindCol()** и **SQLFetch()** для выборки данных из несвязанных столбцов прикладная программа может использовать функцию **SQLGetData()**. Она позволяет выполнить выборку данных из столбцов, для которых область хранения не была

заранее подготовлена с помощью функции **SQLBindCol()**. Функция **SQLGetData()** вызывается после **SQLFetch()** для выборки данных из текущей строки.

Алгоритм программы, использующей **SQLFetch()** и **SQLGetData()**, для извлечения данных из каждой строки результирующего множества предполагает выполнение следующих шагов.

1. Вызывается функция **SQLFetch()** для продвижения на следующую строку.

2. Вызывается функция **SQLGetData()** для каждого столбца, который должен быть возвращен из результирующего множества.

3. Повторяются шаги 1 и 2 до тех пор, пока функция **SQLFetch()** не возвратит **SQL_NO_DATA_FOUND**. Это указывает на то, что был достигнут конец результирующего множества. Если результирующее множество пустое, то **SQL_NO_DATA_FOUND** будет возвращен при первом вызове **SQLFetch()**.

RETCODE SQLGetData (hstmt, icol, fcType, rgbValue, cbValueMax, pcbValue);

HSTMT hstmt; — идентификатор оператора.

UWORD icol; — номер столбца результирующего множества, упорядоченный слева направо, начиная с 1.

SWORD fcType; — С-тип данных результирующего множества.

PTR rgbValue; — указатель области хранения данных. Если rgbValue является нулевым указателем, то драйвер “отвязывает” столбец. Для “отвязывания” всех столбцов прикладная программа вызывает функцию **SQLFreeStmt()** с опцией **SQL_UNBIND**.

SDWORD cbValueMax; — максимальная длина буфера rgbValue. Для символьных данных rgbValue должен также включать в себя место для байта нулевого окончания.

SDWORD FAR* pcbValue; — число байт, которое может возвращаться в rgbValue при вызове **SQLGetData()**

Прикладная программа может использовать **SQLBindCol()** для некоторых столбцов, а **SQLGetData()** – для других столбцов в пределах той же строки.

С использованием функции **SQLGetData()** записанный выше фрагмент доступа к данным будет выглядеть следующим образом:

```
...
SQLExecDirect (dbc, "SELECT name, town FROM S", SQL_NTS);
for (;;)
{
    rc = SQLFetch (stmt);
    if (rc == SQL_NO_DATA_FOUND) break;
    SQLGetData (stmt, 1, SQL_C_CHAR, &name,
                (SDWORD)sizeof(name), &val_name);
    ....
    SQLGetData (stmt, 2, SQL_C_CHAR, &town,
                (SDWORD)sizeof(town), &val_town);
    ...
}
```

Непосредственное выполнение оператора

Переход к очередной строке

Надлежащие проверки

Выбор данных столбца

ODBC обеспечивает богатый набор функций расширенной выборки данных, которые позволяют выполнить дополнительные операции над результирующим множеством. Эти функции предусматривают использование блочных и перемещаемых курсоров. Блочные курсоры позволяют выбирать множество строк за один раз, перемещаемые курсоры предназначены для продвижения вперед и назад в пределах результирующего множества. Одна из таких функций – **SQLExtendedFetch()** – предусматривает использование блочных, перемещаемых и блочных и перемещаемых курсоров одновременно. Эта функция расширяет функциональные возможности **SQLFetch()** по следующим направлениям:

- для каждого связанного столбца возвращает множество строк (одна или более) в виде массива;

- при установке типа перемещения она позволяет передвигаться по результирующему множеству в произвольном направлении;
- функция **SQLExtendedFetch()** работает совместно с функцией **SQLSetStmtOption()**.

При выборке одной строки данных за один раз в прямом направлении достаточно воспользоваться функцией **SQLFetch()**.

RETCODE SQLExtendedFetch (hstmt, fFetchType, irow, pcrow, RowStatus);	
HSTMT stmnt;	– идентификатор оператора;
FFetchType;	– тип выборки;
INT irow;	– число строк выборки;
INT pcrow;	– число реально извлеченных строк;
INT rowStatus;	– массив значений состояний

Функция **SQLSetStmtOption()** носит вспомогательный характер и имеет широкий спектр действий. Она служит, в частности:

- для установления размера строкового множества при использовании блочных курсоров;
- определения типа курсора при использовании перемещаемых курсоров;
- установления типа согласования;
- включения идентификатора записи и т. д.

Полное ее рассмотрение выходит за пределы пособия, поэтому в дальнейшем мы ограничимся лишь примерами ее использования.

Строковые множества из более чем одной строки могут быть получены функцией **SQLExtendedFetch()**. Размер строкового множества устанавливается функцией **SQLSetStmtOption()** со значением fOption SQL_ROWSET_SIZE и опцией vParam.

Перемещаемые курсоры, создаваемые функцией **SQLExtendedFetch()**, являются более гибкими, чем курсоры, обеспечиваемые функцией **SQLFetch()**. Прежде чем использовать перемещаемые курсоры, необходимо выбрать тип курсора с помощью функции **SQLSetStmtOption()** со значением fOption SQL_CURSOR_TYPE. Тип курсора, предполагаемый по умолчанию (SQL_CURSOR_FORWARD_ONLY), не позво-

ляет использовать никакие типы перемещения в функции **SQLExtendedFetch()**, кроме **SQL_FETCH_NEXT**.

Функцией **SQLSetStmtOption()** могут быть определены следующие типы курсоров.

- **SQL_CURSOR_STATIC** – статические курсоры, абсолютно нечувствительные к изменениям в таблицах, на основе которых выбрано результирующее множество. Это означает, что как только создано результирующее множество, оно будет оставаться одним и тем же до тех пор, пока не будет закрыто.

- **SQL_CURSOR_DYNAMIC** – динамические курсоры, полностью чувствительные к основным изменениям. Это означает, что данные результирующего множества будут постоянно изменяться, отражая вставки, обновления и удаления после того как множество было создано.

- **SQL_CURSOR_KEYSET_DRIVEN** – курсоры, управляемые ключами, занимают некоторое среднее положение между динамическими и статическими курсорами с точки зрения того, как они отражают основные изменения в таблице базы данных. Такое результирующее множество содержит обновленные и удаленные строки, но не содержит новых строк. Курсоры данного типа являются наилучшим средством в тех случаях, когда необходимо применять динамическое результирующее множество, но нет возможности использовать динамические курсоры.

Формат функции **SQLSetStmtOption()**, используемой для задания типа курсора, следующий:

RETCODE SQLSetStmtOption (hstmt, SQL_CURSOR_TYPE , fCursor)	
HSTMT stmnt;	– идентификатор оператор;
UDWORD fCursor;	– необходимый курсор

Способность продвигать курсор на любое число строк является реальным преимуществом. Например, если курсор установлен на первую строку результирующего множества и необходимо переместить его на последнюю строку, функция **SQLExtendedFetch()** позволит сделать это за одно обращение, в то время как с использованием функции **SQLFetch()** могут потребоваться тысячи вызовов.

Замечание. Так как **SQLExtendedFetch()** является функцией второго уровня, ее поддерживают не все драйверы. Для того чтобы это

выяснить, необходимо вызвать функцию **SQLGetFunctions()** с параметром `fFunction SQL_API_SQLEXTENDEDFETCH`. Функция возвратит значения `TRUE` либо `FALSE`, что даст ответ на вопрос о поддержке драйвером этой функции.

Характер перемещения курсора устанавливается с помощью параметров `fFetchType` и `irow` в функции **SQLExtendedFetch()**. `fFetchType` определяет тип выполняемой выборки, а `irow` (при его использовании) – число строк выборки.

Приведенная ниже таблица содержит характеристики результирующего множества, получаемого функцией **SQLExtendedFetch()** в зависимости от параметра `fFetchType`.

Значение параметра <code>fFetchType</code>	Действие SQLExtendedFetch()
<code>SQL_FETCH_NEXT</code>	Извлекает следующее строковое множество
<code>SQL_FETCH_PRIOR</code>	Извлекает предыдущее строковое множество
<code>SQL_FETCH_RELATIVE</code>	Извлекает строковое множество, начиная с <i>n</i> -й строки по отношению к текущей позиции курсора, где <i>n</i> определяется параметром <code>irow</code>
<code>SQL_FETCH_FIRST</code>	Извлекает первое строковое множество
<code>SQL_FETCH_LAST</code>	Извлекает последнее строковое множество
<code>SQL_FETCH_ABSOLUTE</code>	Извлекает строковое множество, начиная с <i>n</i> -й строки в результирующем множестве, где <i>n</i> определяется параметром <code>irow</code>

Таким образом, для выборки результирующих данных используется функция **SQLFetch()**, которая за один раз продвигает курсор только вперед и только на одну строку и действует до тех пор, пока не будет возвращено `SQL_NO_DATA_FOUND`. Недостатком такой выборки является то, что, например, при необходимости переместиться с первой строки результирующего множества на последнюю функция **SQLFetch()** будет последовательно проходить по всем строкам выборки. Более эффективно в этом случае использование функцией **SQLExtendedFetch()** перемещаемых курсоров, которые способны перемещаться в любом направлении и на любое количество строк. Однако эта функция есть функция второго уровня, и ее поддерживают не все драйверы.

2.8. НАСТРОЙКА ODBC

Для использования средств ODBC на платформе Unix необходимо:

- установить программное обеспечение ODBC и драйверы для соответствующих СУБД;
- выполнить системные требования для подключения и работы с заданным сервером баз данных;
- подключить к исходной программе необходимые заголовочные файлы;
- при компиляции программы подключить необходимые библиотеки;
- определить источники данных пользователя в системном файле `odbc.ini`.

Программное обеспечение ODBC включает системные библиотеки и заголовочные файлы, предоставляющие прикладной программный интерфейс для взаимодействия программ с различными СУБД, а также утилиты командной строки для настройки и управления источниками данных. В операционных системах на платформе Unix используется пакет с открытым исходным кодом `unixODBC`, который реализует подсистему ODBC. Драйверы ODBC для подключения к различным системам управления базами данных распространяются в виде пакетов и библиотек (`postgresql-odbc`, `mysql-connector-odbc`, `oracle-instantclient-odbc` и пр.) как отдельно, так и в составе дистрибутивов самих СУБД.

Системные требования. Системные требования зависят от конкретной СУБД, с которой работает пользователь. Как правило, выполнение требований обеспечивается установкой некоторых утилит и заданием переменных окружения в интерактивном сеансе пользователя. Переменные окружения пользователя для командной оболочки Bash задаются в конфигурационном файле `~/.bashrc` в домашней директории, однако также часто могут быть установлены системным администратором для всей системы целиком. В переменных окружения задаются пути к директориям с конфигурационными и исполняемыми файлами, имена серверов, баз данных, настройки локалей и пр.

Типовые установки переменных окружения для СУБД Informix:

```
export INFORMIXSERVER=fpm
export INFORMIXDIR=/usr/Informix
export INFORMIXSQLHOSTS=${INFORMIXDIR}/etc/sqlhosts
export PATH=${INFORMIXDIR}/bin:${PATH}
export ONCONFIG=onconfig
export DB_LOCALE=ru_RU.UTF-8
export CLIENT_LOCALE=ru_RU.UTF-8
```

Типовые установки переменных окружения для СУБД Oracle:

```
export ORACLE_HOSTNAME=fpm
export ORACLE_TERM=xterm
export ORACLE_SID=students
export
ORACLE_HOME=/opt/oracle/app/oracle/product/11.2.0/dbhome_1
export PATH=${ORACLE_HOME}/bin:${PATH}
export NLS_LANG=RUSSIAN_CIS.AL32UTF8
export NLS_DATE_FORMAT="DD-MON-YYYY HH24:MI:SS"
```

При работе с PostgreSQL каких-либо настроек на уровне системы делать не требуется, однако с помощью установки переменных окружения PG* (PGHOST, PGPORT, PGDATABASE, PGUSER, PGOPTIONS и пр.) можно изменить параметры по умолчанию и поведение клиентских программ.

Заголовочные файлы. Для получения доступа к ODBC-функциям в программе должны быть описаны заголовочный файл `sqltext.h` и файл `sqltypes.h`, где находятся описания типов данных, используемых в ODBC.

Компиляция исходной программы. Как и любой исходный файл, написанный на языке C, файл с программой, вызывающей ODBC-функции, должен иметь расширение `.c`. При компиляции необходимо подключить необходимые библиотеки, используемые в вызываемых

функциях. Поскольку в этом случае командная строка принимает достаточно сложный вид, целесообразно на уровне системного администратора оформить командный файл, в котором подключаются необходимые библиотеки.

Командный файл `odbcc`, используемый для трансляции и компоновки программы, может иметь такой вид:

```
#!/bin/bash
CC=/usr/bin/gcc
ODBCPATH=/usr
LFLAGS="-L${ODBCPATH}/lib64"
CFLAGS="-I${ODBCPATH}/include"
LIBS="-lodbc"
${CC} ${LIBS} ${CFLAGS} ${LFLAGS} -o "${1%.[^.]*}.exe" "${@}"
```

С использованием такого файла компиляция выполняется с помощью простой команды, дающей на выходе исполняемый файл:

`odbcc <имя файла>.c`

Системный файл `odbc.ini` представляет собой текстовый конфигурационный файл, который содержит список источников данных DSN (Data Source Name) и специфичные настройки для каждого источника. Существует два типа файла `odbc.ini` – с общими настройками для всей системы, который обычно располагается в `/etc/odbc.ini`, и с пользовательскими настройками, который создается в домашней директории пользователя `~/odbc.ini`. В дополнение к стандартным путям расположения файла существует переменная окружения `ODBCINI`, с помощью которой можно переопределить путь к пользовательскому файлу `.odbc.ini`, если он находится не в домашнем каталоге. Переменную окружения `ODBCINI` для командной оболочки Bash можно указать в файле `~/.bashrc` в домашнем каталоге или задать с помощью следующей команды:

```
export ODBCINI=/opt/odbc/usr/local/etc/.odbc.ini
```

По умолчанию диспетчер драйверов при поиске источников данных использует следующую схему поиска файлов:

- 1) если в пользовательском сеансе установлена переменная окружения ODBCINI – используется файл, указанный в этой переменной;
- 2) если в домашнем каталоге пользователя существует файл *.odbc.ini* – используется пользовательский файл;
- 3) в случае отсутствия пользовательских файлов источников данных используется общесистемный файл */etc/odbc.ini*.

Для настройки пользовательских источников данных необходимо создать текстовый файл *.odbc.ini* в домашней директории пользователя и отредактировать его. В файле должен быть под некоторым идентификатором описан требуемый источник данных, имя которого используется функцией *SQLConnect()*, и должен присутствовать раздел с данным именем, в котором содержатся атрибуты, описывающие источник данных. Например, если в качестве источника данных используется база данных PostgreSQL, описание может быть следующим:

```
[MYDB]
Description = PostgreSQL connection to Students
Driver = PostgreSQL
Database = students
Servername = students.ami.nstu.ru
UserName =
Password =
Port = 5432
Protocol = 9.3
Trace = Yes
TraceFile = sql.log
ReadOnly = No
RowVersioning = No
ShowSystemTables = No
ShowOidColumn = No
FakeOidIndex = No
ConnSettings =
```


В приведенном фрагменте MYDB – имя источника данных, параметр Driver содержит имя драйвера, обслуживающего источник данных – PostgreSQL. Соответствующие библиотеки средств ODBC для СУБД PostgreSQL должны быть установлены в системе, а сам драйвер прописан в системном файле /etc/odbcinst.ini:

```
[PostgreSQL]
Description = ODBC for PostgreSQL
Driver = /usr/pgsql-9.3/lib/psqlodbc.so
Setup = /usr/lib64/libodbcpsqlS.so
Driver64 = /usr/pgsql-9.3/lib/psqlodbc.so
Setup64 = /usr/lib64/libodbcpsqlS.so
FileUsage = 1
```

Путь к системному файлу odbcinst.ini и имя самого файла могут быть переопределены с помощью системных переменных окружения ODBCYSINI и ODBCINSTINI соответственно. Полный список параметров описания источника данных в файле .odbc.ini приведен ниже в таблице.

Атрибуты строки соединения

Атрибут	Описание
ApplicationUsingThreads (AUT)	ApplicationUsingThreads={0 1}. Обеспечивает работу драйверов с многопоточковыми приложениями
CancelDetectInterval (CDI)	Позволяет прерывать долго выполняющиеся запросы в потоковых приложениях. Выбирает значение, определяющее, как часто драйвер будет проверять, был ли запрос закрыт функцией SQLCancel(). Например, если CDI = 5, то для каждого затянувшегося запроса драйвер каждые 5 с будет проверять, прервал ли пользователь выполнение запроса, используя функцию SQLCancel(). Значение по умолчанию, равное 0, свидетельствует о том, что запросы не будут прекращены до тех пор, пока не завершат свое выполнение

Продолжение таблицы

Атрибут	Описание
CursorBehavior (CB)	CursorBehavior = {0 1}. Этот атрибут определяет, будет ли курсор сохранен или закрыт по завершении каждой транзакции. Значение по умолчанию равно 0 (курсor закрывается). Значение атрибута, равное 1, означает, что курсор будет сохранен на текущей позиции по завершении транзакции. Значение CursorBehavior = 1 может влиять на выполнение операций при работе с базой данных
Database (DB)	Имя базы данных, к которой необходимо получить доступ
DataSourceName (DSN)	Строка, которая определяет источник данных в системной информации
EnableInsertCursors (EIC)	EnableInsertCursors = {0 1}. Определяет, будет ли драйвер использовать вставку курсоров. Увеличивает быстродействие в течение многократных вставок. Позволяет буферизовать вставляемые данные в памяти, прежде чем записать на диск. Когда установлено, что EnableInsertCursors = 0, то драйвер не использует вставку курсоров
EnableScrollable Cursors (ESC)	EnableScrollableCursors = {0 1}. Атрибут определяет, <u>обеспечивает ли драйвер</u> вставку скроллирующих курсоров. Значение по умолчанию равно 0 (без использования скроллирующих курсоров). Если установлена поддержка скроллирующих курсоров (EnableScrollableCursors = 1), то в результирующем множестве не должно быть слишком длинных столбцов (SQL_LONGVARCHAR или SQLVARBINARY)
GetDBListFromPostgreSQL (GDBLFI)	GetDBListFromPostgreSQL={0 1}. Этот атрибут определяет, будет ли драйвер запрашивать список базы данных от сервера PostgreSQL или от пользователя, который определил его при установке драйвера. Значение по умолчанию равно 1, т. е. драйвер запрашивает список от сервера
HostName (HOST)	Имя машины, на которой установлен сервер PostgreSQL

Окончание таблицы

Атрибут	Описание
LogonID (UID)	Ваше имя, указанное на сервере PostgreSQL
Password (PWD)	Пароль
ServerName (SRVR)	Имя сервера, который обрабатывает базу данных
Service (SERV)	Имя службы, которое установлено на главном компьютере системным администратором
UseDefaultLogin (UDL)	UseDefaultLogin = {0 1}. Значение 1 позволяет считывать имя и пароль непосредственно из системного реестра PostgreSQL. Значение по умолчанию равно 0

В таблице даются полные и сокращенные названия каждого атрибута, а также их описание. В системном файле используются только полные названия атрибутов. Значения заданы по умолчанию и используются в таком виде, если нет других значений, заданных в строке соединения или системном файле. Если изменить значение атрибута при настройке источника данных, то это значение будет использоваться по умолчанию.

Отметим, что в файле `.odbc.ini` может быть описано несколько источников данных (разные базы данных, разные СУБД) и пользовательская программа может одновременно работать с разными источниками данных.

3. ДОСТУП К БАЗАМ ДАННЫМ ПОСРЕДСТВОМ CGI-СКРИПТОВ

В данном разделе излагаются основные понятия программирования CGI-скриптов и способы написания на их основе программ доступа к базам данных. Поскольку эта тема напрямую затрагивает работу WWW-сервера, желательно предварительно ознакомиться со средствами создания HTML-документов. Так как средства HTML являются достаточно обширными, то в дальнейшем мы ограничимся лишь теми из них, которые непосредственно связаны с темой настоящего пособия.

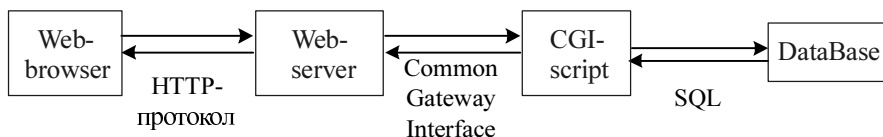


Рис. 3.1. Доступ к базе данных из WWW-среды посредством CGI-скрипта

Общая схема доступа к базам данных в этом случае может быть представлена в виде диаграммы, показанной на рис. 3.1.

3.1. ПОНЯТИЕ CGI-СКРИПТА

Связь клиент-сервер, имеющая место между программой-браузером и Web-сервером, очень хорошо проявляет себя при обслуживании HTML и графических файлов из каталогов Web-сервера. Но помимо доступа к статическим документам сервера существует потребность получения документов как результата выполнения прикладной программы. Получение документов как результата выполнения прикладной программы (CGI-скрипта) реализуется на сервере WWW благодаря использованию общего стандартного интерфейса CGI (Common Gateway Interface).

Спецификация CGI описывает формат и правила обмена данными между программным обеспечением WWW-сервера и запускаемой программой и представляет собой общую среду и набор протоколов для внешних приложений, которые используются при взаимодействии с Web-сервером. При осуществлении таких соединений сервер отвечает за получение информации от программы-браузера и пересылку данных обратно на браузер. CGI-программа (CGI-скрипт) представляет собой программу локальной операционной системы сервера (в двоичном виде или в виде программы для интерпретатора), которая может быть вызвана из среды WWW. Для инициирования CGI достаточно, чтобы в URL-адресе был указан путь до запускаемой программы. Программное обеспечение WWW-сервера исполняет эту программу, передает ей входные параметры и возвращает результаты ее работы как результат обработки запроса клиенту. Это существенно расширяет диапазон функций Web-сервера. CGI-программа может обрабатывать сигналы с датчиков установок, взаимодействовать с мощным сервером баз данных и т. п.

С целью облегчения администрирования CGI-программ, а также для удовлетворения требованиям безопасности CGI-программы группируются в одном или нескольких явно указанных серверу каталогах. По умолчанию это каталог `cgi-bin` в иерархии серверных каталогов, однако его имя и положение могут различаться, но в таком случае большинство Web-серверов требуют специальной дополнительной настройки. Популярный Web-сервер Apache с модулем `UserDir` позволяет пользователям сервера размещать страницы и CGI-скрипты без выделения им доступа к корневой директории Web-сервера. Статические объекты и динамические веб-страницы размещаются в каталоге `~/public_html`, а CGI-скрипты в каталоге `~/public_html/cgi-bin`. Поскольку файл CGI-программы, помещенный в каталог `cgi-bin`, при обращении будет вызван и непосредственно исполнен Web-сервером, то необходимо, чтобы:

- в настройках Web-сервера были разрешены вызовы обработчиков CGI и явно определены каталоги `cgi-bin` в иерархии серверных каталогов;
- права доступа к каталогам `cgi-bin`, в которых хранятся CGI-скрипты, были достаточными для доступа к ним Web-сервера (кроме того, такие же права должны быть и у файлов, к которым обращается сам Web-сервер напрямую или посредством CGI-скриптов);
- файлы CGI-скриптов были обязательно исполняемыми.

Ниже приведен пример команд для создания необходимой структуры каталогов Web-сервера в домашней директории пользователя и размещения CGI-скрипта first.cgi:

```
mkdir ~/public_html
mkdir ~/public_html/cgi-bin
cp first.cgi ~/public_html/cgi-bin/first.cgi
chmod a+rx ~
chmod a+rx ~/public_html
chmod a+rx ~/public_html/cgi-bin
chmod a+rx ~/public_html/cgi-bin/first.cgi
```

В общем случае, CGI-программами могут быть как скомпилированные двоичные файлы, так и текстовые скрипты на различных интерпретируемых языках, таких как Bash, Perl, Python, а соответствующие им расширения файлов - *.cgi, *.bin, *.exe, *.sh, *.pl, *.py, *.рус. В последующем изложении будет рассматриваться CGI-программирование на языке C, поскольку в данном случае важно рассмотреть сам принцип CGI-программирования, а его реализация на различных языках мало различается и не имеет большого значения.

Схема действий выглядит следующим образом: пишется программа на C, затем она компилируется, исполняемый файл с расширением *.cgi помещается в каталог для CGI-скриптов в домашней директории пользователя. Данная процедура может различаться для различных платформ сервера. Для доступа к CGI-скрипту, расположенному в cgi-bin директории пользователя, клиенту следует использовать URL: <http://students.ami.nstu.ru/~pmxxyy/cgi-bin/first.cgi>.

Ниже приведен пример простейшей CGI-программы:

```
#include <stdio.h>
int main(void) {
    printf("Content-Type: text/html;charset=utf-8\n\n");
    printf("<HTML><BODY>");
    printf("<H1>Hello world!!!</H1>");
    printf("</BODY></HTML>");
    return 0;
}
```

Строка `printf(«Content-Type:text/html; charset=utf-8\n\n»);` является заголовком, указывающим тип содержимого. В данном случае тип генерируемого документа – HTML в кодировке UTF-8, хотя может быть и любым другим. Тип содержимого формируется из двух составляющих: базового типа и подтипа (формата). Сначала указывается один из базовых типов: `application` – данные для какого-либо приложения; `audio` – аудиоданные, например `RealAudio`; `video` – видеоданные, например `MPEG` или `AVI`; `image` – изображение; `text` – текст; далее через слэш (/) указывается подтип (формат). Примеры заголовков:

- `Content-type: application/javascript`
- `Content-type: text/html`
- `Content-type: text/plain`
- `Content-type: image/gif`
- `Content-type: video/mpeg`

Еще заголовок может включать в себя вспомогательные параметры, `Content-Length` – длина содержимого, `Expires` – время существования в кэше и т. д. Каждый параметр указывается в отдельной строке и заканчивается переносом строки `\n`.

Кроме `Content-Type` возможно поле `Location`, содержащее URL ресурса, на который скрипт перенаправляет запрос. К примеру, вместо `Content-Type: text/html` могло бы стоять:

`Location: http://www.example.com/index.html`

Возможное поле `Status` позволяет вернуть статус обработки (код ошибки). Информация о коде находится в строке состояния при ответе сервера. Коды возврата протокола HTTP находятся в приложении 3.

После того как заголовок сформирован, надо отправить пустую строку, для того чтобы отделить его от основных данных HTML – два символа `\n\n` образуют пустую строку. Все, что скрипт печатает в стандартный поток вывода `STDOUT`, идет на обработку Web-серверу, который формирует ответ и посылает его программе-браузеру. В нашем случае браузер получает страницу с HTML-кодом:

`<HTML><BODY><H1>Hello world!!!</H1></BODY></HTML>`

В приведенном примере отсутствует обмен данными. Прежде чем приступить к описанию механизма, посредством которого пользова-

тель вводит определенные данные и в результате работы скрипта получает на них ответ, необходимо ознакомиться с программированием HTML-формы.

3.2. ПОНЯТИЕ HTML-ФОРМЫ

В общем случае интерактивный интерфейс пользователя представляет собой систему, обеспечивающую взаимодействие пользователя и программы. Для WWW интерактивный интерфейс можно определить как последовательность HTML-документов, реализующих интерфейс пользователя. Можно также условно классифицировать принципы построения интерфейса по типу формирования HTML-документа на статический и динамический. В первом случае источником интерфейса является HTML-документ, созданный в каком-либо текстовом или HTML-ориентированном редакторе. Во втором случае источником интерфейса является HTML-документ, сгенерированный cgi-модулем, что обеспечивает гибкость в видоизменении интерфейса во время использования. Задача построения интерфейсов разделяется на клиентскую и серверную части. Для создания клиентской части необходимо иметь HTML-документ, в котором реализован интерфейс с пользователем. Для успешного решения подобной задачи HTML-документы должны содержать некоторые элементы управления. Под элементами управления понимаются флажки-переключатели, радио- и обычные кнопки, поля ввода текста (одно- и многострочные), списки с единственным и множественным выбором и т. д. Все подобные элементы можно создавать с помощью различных тэгов HTML. При этом каждый такой элемент управления должен принадлежать объекту HTML, носящему название *формы*. Иными словами, форма представляет собой контейнер, в котором хранятся объекты управления.

Если целью формы является сбор данных для последующей передачи их серверу, то такая форма должна непременно содержать следующее.

1. Адрес CGI-скрипта (программы-обработчика, расположенной на некотором сервере), которому будут пересылаться данные из формы. При этом пересылается не вся страница, а только значения, соответствующие элементам управления формы, которая инициировала запуск программы-обработчика.

2. Метод передачи данных (наиболее применяемые POST и GET).

3. Некоторый объект формы, при нажатии на который произойдет пересылка данных. Очень часто в качестве такого объекта использует-

ся кнопка со специальным значением «Submit», которая отвечает за отправку данных из формы серверу.

Форма задается в HTML-документе с помощью тэга FORM. Все элементы управления находятся внутри контейнера <FORM>...</FORM>.

```
<FORM action="http://.....cgi" method="GET"|“POST”
  enctype="encodingType" name="formName" target="windowName”
  onSubmit="Handler">
    ...          Поля формы          ...
</FORM>
```

Тэг FORM имеет ряд атрибутов. Среди них для CGI-программирования наиболее важны первые два (ими можно ограничиться на начальной стадии написания CGI-программ):

- action – задает URL-адрес программы (CGI-скрипта), которая будет обрабатывать данные формы. Если он опущен, используется URL-адрес текущего документа;
- method – задается метод. По умолчанию предполагается GET.

Далее приведен пример простой формы, которая осуществляет сбор данных и отправляет их серверу для обработки.

Пример использования текстового поля и радиокнопки в форме:

```
<HTML>
<HEAD>
<TITLE>Пример использования CGI</TITLE> </HEAD>
<BODY>
<FORM ACTION="http://127.0.0.1/cgi-bin/test.cgi”
METHOD="POST">
<B>Введите имя<I>(Фамилию Имя Отчество)</I>:</B>
<INPUT name=RealName type=text size=40 maxlength=60 value=
“Иванов Иван Иванович”><P>
Пол: <INPUT name=Sex type=Radio value="1” CHECKED>- мужской
    <INPUT name=Sex type=Radio value="2”> - женский<P>
<INPUT name=Submit type=submit value="Послать запрос”><BR>
<INPUT name=Reset type=reset value="Сброс”>
</FORM> </BODY> </HTML>
```

Более сложные элементы HTML-формы позволяют задавать:

- поля ввода пароля;
- большие текстовые многострочные области для ввода больших текстов;
- переключатели с зависимой и независимой фиксацией состояния;
- ниспадающие списки;
- списки с множественным выбором вариантов и пр.

```
<HTML><HEAD>
<TITLE>Пример страницы с Web-формой</TITLE>
</HEAD>
<BODY><FORM>
<TABLE>
<TR><TD colspan=2>
Пример формы, демонстрирующей основные элементы управле-
ния:<BR>
</TD></TR>
<TR><TD>Текстовые поля ввода
<INPUT name="test_text" value="Текстовое поле"></TD>
<TD>Поля ввода пароля
<INPUT type="password" name="test_pass"
value="MyPassword"></TD>
</TR>
</TABLE><BR>
<TABLE>
<TR><TD>Большие текстовые многострочные области<BR>
для ввода больших объемов текста<BR>
<TEXTAREA name="text_textarea" rows=4 cols=20>
Текстовая область</TEXTAREA>
```

```

</TD>
<TD>Переключатели с зависимой<BR>
<INPUT type="radio" name="test_radio" value="1"
checked>Радиокнопка 1
<INPUT type="radio" name="test_radio" value="2">Радиокнопка 2
<INPUT type="radio" name="test_radio" value="3">Радиокнопка 3
<INPUT type="radio" name="test_radio" value="4">Радиокнопка 4
<BR>
и независимой фиксации состояния<BR>
<INPUT type="checkbox" name="test_checkbox" value="1">Флажок 1
<INPUT type="checkbox" name="test_checkbox" value="2">Флажок 2
<INPUT type="checkbox" name="test_checkbox" value="3">Флажок 3
<INPUT type="checkbox" name="test_checkbox" value="4">Флажок 4
</TD>
</TR>
</TABLE>
<TABLE>
<TR align=center><TD>Выпадающие списки
    <SELECT name="test select">
        <OPTION value="1" selected>Вариант 1
        <OPTION value="2" >Вариант 2
        <OPTION value="3" >Вариант 3
        <OPTION value="4" >Вариант 4
        <OPTION value="5" >Вариант 5
        <OPTION value="6" >Вариант 6
        <OPTION value="7" >Вариант 7

```

```

        </SELECT><BR>
    </TD>
    <TD> Списки с <BR>
множественным выбором вариантов<BR>
    <SELECT name="test selectmultiple" multiple size=4>
        <OPTION value="1" selected>Вариант 1
    <OPTION value="2" >Вариант 2
        <OPTION value="3" >Вариант 3
    <OPTION value="4" selected>Вариант 4
        <OPTION value="5" >Вариант 5
    <OPTION value="6" >Вариант 6
        <OPTION value="7" selected>Вариант 7
    </SELECT>
    </TD>
    <TD>Для удобства передачи внутренней<BR>
информации в формах<BR>
предусмотрены скрытые поля.
        <INPUT type="hidden" name="test_hidden" value="myhidden">
    </TD>
</TR></TABLE>
<TABLE>
<TR><TD>После того как пользователь ввел данные,
он нажимает кнопку передачи данных.
<INPUT type="submit" value "Подача запроса"></TD>
</TR>

```

Окончание фрагмента программы

```
<TR><TD>На случай неправильного ввода в форме можно  
предусмотреть кнопку сброса к значениям, установленным  
по умолчанию. <INPUT type="reset" value "Сброс"></TD>  
</TR> </TABLE>  
</FORM></BODY></HTML>
```

На рис. 3.2 показан вид страницы, в котором приведенная выше HTML-форма отображается на экране.

Пример страницы с Web-формой

Пример формы, демонстрирующей основные элементы управления:

Текстовые поля ввода Поля ввода пароля

Большие текстовые многострочные
области для ввода
больших объемов текста

Текстовая область

▲

▼

Переключатели с зависимой
● Радиокнопка 1 ○ Радиокнопка 2
○ Радиокнопка 3 ○ Радиокнопка 4
и независимой фиксацией состояния
☐ Флажок 1 ☐ Флажок 2 ☐ Флажок 3

Списки

с множественным выбором

Выпадающие списки

Вариант 1	▼
Вариант 2	
Вариант 3	
Вариант 4	

вариантов

Вариант 1

Вариант 2

Вариант 3

Вариант 4

Вариант 5

▲

▼

Для удобства передачи
внутренней информации
в формах предусмотрены
скрытые поля.

После того как пользователь ввел данные, он нажимает кнопку пере-
дачи данных.

На случай неправильного ввода в форме можно предусмотреть кноп-
ку сброса к значениям, установленным по умолчанию

Рис. 3.2. Результаты работы HTML-формы

3.3. ПЕРЕМЕННЫЕ CGI-ОКРУЖЕНИЯ

Сервер при запуске CGI-скрипта формирует среду окружения, в которой скрипт может найти всю доступную информацию о HTTP-соединении и полученных в запросе параметрах. Большинство переменных стандартизированы.

- **SERVER_SOFTWARE** содержит информацию о WWW-сервере (название/версия).

Apache/1.0

- **SERVER_NAME** содержит информацию об имени машины, на которой запущен WWW сервер, символическое имя или IP-адрес, соответствующие URL.

SERVER_NAME=www.lupa.ru

- **GATEWAY_INTERFACE** содержит информацию о версии CGI (CGI/версия).

GATEWAY_INTERFACE=CGI/1.1

- **CONTENT_LENGTH** – значение этой переменной соответствует длине стандартного входного потока в символах.

CONTENT_LENGTH=41

- **CONTENT_TYPE** содержит тип тела запроса.

- **SERVER_PROTOCOL** – эта переменная содержит информацию об имени и версии информационного протокола (протокол/версия).

SERVER_PROTOCOL=HTTP/1.1

- **SERVER_PORT** – значение переменной содержит номер порта, на который был послан запрос.

SERVER_PORT=80

- **REQUEST_METHOD** – метод запроса, который был использован (POST, GET, HEAD и т.д.).

REQUEST_METHOD=GET

- **PATH_INFO** – значение переменной содержит полученный от клиента виртуальный путь до cgi-модуля.

- **PATH_TRANSLATED** – значение переменной содержит физический путь до cgi-модуля, преобразованный из значения PATH_INFO.

- **SCRIPT_NAME** – виртуальный путь к исполняемому модулю, используемый для получения URL.

SCRIPT_NAME=/~paa/script.cgi

- **QUERY_STRING** – значение этой переменной соответствует строке символов, следующей за знаком «?» в URL. Эта информация не декодируется сервером.

QUERY_STRING= name=Metos&age=30

– **REMOTE_HOST** содержит символическое имя удаленной машины, с которой был произведен запрос. В случае отсутствия данной информации сервер присваивает пустое значение и устанавливает переменную **REMOTE_ADDRESS**.

REMOTE_HOST=soft.com

– **REMOTE_ADDRESS** содержит IP-адрес клиента.

REMOTE_ADDRESS=121.172.25.123

– **AUTH_TYPE** – если WWW-сервер поддерживает аутентификацию (подтверждение подлинности) пользователей и cgi-модуль защищен от постороннего доступа, то значение переменной специфицирует метод аутотентификации.

– **REMOTE_USER** содержит имя пользователя в случае аутотентификации.

– **REMOTE_IDENT** содержит имя пользователя, полученное от сервера (если сервер поддерживает аутентификацию согласно RFC 931).

– **HTTP_ACCEPT** – список типов MIME, известных клиенту. Каждый тип в списке должен быть отделен запятой согласно спецификации HTTP (тип/подтип,тип/подтип и т. д.)

– **HTTP_USER_AGENT** – название программы просмотра, которую использует клиент при отправке запроса.

Ниже приведен скрипт, который выводит переменные окружения.

```
#include<stdio.h>
#include<stdlib.h>
int main(int arg,char *argv[])
{
    char *request_method;
    char *query_string;
    char *content_length, type;
    char *remote_addr;
    char *server_name;
    char *script_name, filename;
    char *path_info, translated;
    char *http_user_agent;
    char *no_variable="переменная не определена";
```

```
printf("Content-Type: text/html\n\n");
printf("<HTML>\n");
printf("<HEAD><TITLE>VAR CGI Example</TITLE></HEAD>\n");
printf("<BODY>\n");
printf("<CENTER><H1>Пример CGI на C</H1></CENTER>");
printf("<HR>\n"); printf("<H3>Переменные окружения:</H3>\n");
// Получаем переменные
request_method=getenv("REQUEST_METHOD");
    if (!request_method) request_method=no_variable;
query_string=getenv("QUERY_STRING");
    if (!query_string) query_string=no_variable;
content_length=getenv("CONTENT_LENGTH");
    if (!content_length) content_length=no_variable;
content_type=getenv("CONTENT_TYPE");
    if (!content_type) content_type=no_variable;
remote_addr=getenv("REMOTE_ADDR");
    if (!remote_addr) remote_addr=no_variable;
server_name=getenv("SERVER_NAME");
    if (!server_name) server_name=no_variable;
script_name=getenv("SCRIPT_NAME");
    if (!script_name) script_name=no_variable;
script_filename=getenv("SCRIPT_FILENAME");
    if (!script_filename) script_filename=no_variable;
path_info=getenv("PATH_INFO");
    if (!path_info) path_info=no_variable;
path_translated=getenv("PATH_TRANSLATED");
    if (!path_translated) path_translated=no_variable;
```

Получение
переменных


```

http_user_agent=getenv("HTTP_USER_AGENT");
    if (!http_user_agent) http_user_agent=no_variable;
// Выведем значения
    printf("REQUEST_METHOD =%s\n<BR>", request_method);
    printf("QUERY_STRING =%s\n<BR>", query_string);
    printf("CONTENT_LENGTH =%s\n<BR>", content_length);
    printf("CONTENT_TYPE =%s\n<BR>", content_type);
    printf("REMOTE_ADDR =%s\n<BR>", remote_addr);
    printf("SERVER_NAME =%s\n<BR>", server_name);
    printf("SCRIPT_NAME =%s\n<BR>", script_name);
    printf("SCRIPT_FILENAME =%s\n<BR>", script_filename);
    printf("PATH_INFO =%s\n<BR>", path_info);
    printf("PATH_TRANSLATED =%s\n<BR>", path_translated);
    printf("HTTP_USER_AGENT =%s\n<BR>", http_user_agent);
    printf("<HR>\n"); printf("</BODY></HTML>\n");
}

```

Вывод значений

3.4. ОБРАБОТКА ФОРМЫ С ПОМОЩЬЮ CGI-СКРИПТОВ

При нажатии некоторой кнопки на форме HTML-документа происходит передача данных браузером серверу, на котором находится программа-обработчик данных. Эта программа получила название CGI-скрипта (так как при передаче данных используется CGI-технология).

CGI-программа обеспечивает:

- получение данных формы HTML-документа в виде строки, в которой перечислены значения всех элементов HTML-формы, инициировавшей запуск программы (данная строка записана в соответствии с четко определенным форматом);
- разбор полученной строки;

- обработку данных (например, занесение полученных значений в базу данных или выборку некоторых записей из базы данных по полученным значениям и т. д.);

- формирование HTML-документа, который будет передан сервером программе-браузеру.

На входе скрипта имеются данные формы, закодированные методом `urlencode`. Кроме того, скрипт получает от сервера некоторые параметры через переменные среды окружения, которые формируются перед запуском скрипта. В зависимости от метода (`GET` или `POST`) данные формы будут помещены в переменную окружения `QUERY_STRING` или поданы на стандартный ввод `STDIN`. Узнать, какой метод применил пользователь, можно, проанализировав переменную `REQUEST_METHOD`. Данные, введенные в форме, передаются CGI-модулю в формате:

имя=значение&имя1=значение1&...&имяN=значениеN

,

где `имяi` – значение параметра `name` из элемента HTML-формы, `значениеi` – введенное или выбранное значение независимо от его типа.

Алгоритм обработки передаваемой в CGI-скрипт строки данных состоит в разделении ее на пары `имя=значение` и декодировании каждой пары. При этом учитывается, что все пробелы во введенных значениях в форме сервером были заменены символом «+» и символы с десятичным кодом больше 128 преобразованы в символ «%» и следующим за ним шестнадцатеричным кодом символа.

Так, приведенная ранее форма из примера 3.1 в ответ на ввод данных

Введите имя (Фамилию, Имя, Отчество):

Пол ☐ мужской ☒ женский

Послать запрос

Сброс

передаст в CGI-скрипт последовательность

```
RealName=Smith&Sex=2&Submit=%F0%CF%D3%CC%C1%D4%D8+
%DA%C1%D0%D2%CF%D3
```

При отправке значений элементов формы по умолчанию

Введите имя (Фамилию, Имя, Отчество):	Иванов Иван Иванович
Пол <input checked="" type="radio"/> мужской <input type="radio"/> Женский	
<input type="button" value="Послать запрос"/>	
<input type="button" value="Сброс"/>	

в CGI-скрипт будет передана последовательность

```
RealName= %E9%D7%C1%CE%CF%D7+%E9%D7%C1%CE+
%E9%D7%C1%CE%CF%D7%C9%DE &Sex=1&
Submit=%F0%CF%D3%CC%C1%D4%D8+%DA%C1%D0%D2%CF
%D3
```

Для реализации взаимодействия клиент–сервер важно, какой метод HTTP-запроса использует клиентская часть при обращении к WWW-серверу. В общем случае запрос – это сообщение, посылаемое клиентом серверу. Первая строка HTTP-запроса определяет метод, который должен быть применен к запрашиваемому ресурсу, идентификатор ресурса (URI-Uniform Resource Identifier) и используемую версию HTTP-протокола. Метод, указанный в строке запроса, может быть одним из следующих.

1. GET. Служит для извлечения информации по заданной в URL-ссылке. При использовании метода GET информация, введенная пользователем, посылается на сервер с помощью переменной окружения QUERY_STRING. Таким образом, длина сообщений при использовании этого метода ограничена. Метод GET используется по умолчанию при описании формы. Данные добавляются в конец URL-адреса и отделяются знаком «?».

`http://127.0.0.1/cgi-bin/varcgi.cgi? RealName=Jack&age=25`

2. HEAD. Практически идентичен GET, но сервер не возвращает тело объекта. В ответе содержится только заголовок. Метод может использоваться для проверки гиперссылок.

3. POST. Передает данные для обработки CGI-программе, указанной в URL. С помощью метода POST информация, введенная пользователем, посылается непосредственно в сценарий с помощью выходного потока сервера STDOUT и входного потока STDIN вашего сценария. Преимущества использования этого метода – неограниченность длины передаваемого сообщения и безопасность передачи по сравнению с GET.

Существуют и другие реже используемые методы.

Проблема перекодировки строки не является слишком сложной. Она подразумевает решение довольно простой задачи перевода числа из одной системы счисления в другую (из шестнадцатеричной – в десятичную). Необходимо учитывать также, что все пробелы при декодировании сервером были заменены символом «+», а все символы с десятичным кодом больше 128 преобразованы в символ «%» и следующий за ним шестнадцатеричный код символа.

Процедура значительно облегчается в случае, если вводимые через форму значения заведомо являются символами с кодом меньше 128. Фрагмент приведенного ниже CGI-скрипта test.cgi, указанного в атрибуте action формы, обрабатывает данные, введенные пользователем для случая, когда текстовое поле с именем содержит лишь фамилию, записанную английскими буквами.

```

...
char String[50],vspom[10];
int Select, j ,i;

...
printf ("Content-Type: text/html\n\n"); printf("<HTML><BODY>");

// Получение строки символов выданной методом POST
i=0;
while ((Value=getc(stdin))!=EOF) { String[i]=Value; i+=1; };
String[i]=0;
// Выделение первого параметра
i=0;
while (String[i] != '=') i+=1;
i+=1; j=0;
while (String[i] != '&') { vspom[j]=String[i]; i+=1; j+=1; };
vspom[j]=0;
// Выделение второго параметра
i++;
while (String[i] != '=') i+=1;
i+=1; j=0;
while (String[i] != '&') { vspom[j]=String[i]; i+=1; j+=1; };
vspom[j]=0;
Select=atoi(vspom);
...
printf("</BODY></HTML>");

```

Строка String содержит последовательность символов, переданных через форму

Строка vspom содержит первый переданный параметр (фамилия)

Строка vspom содержит второй переданный параметр в текстовом виде (номер радиокнопки)

Преобразование строки Select в число

При использовании метода GET разница в обработке данных формы будет заключаться в том, что передаваемую последовательность символов следует брать не из входного потока, как показано в предыдущем примере, а из переменной CGI-окружения QUERY_STRING.

<pre>char *query_string; query_string = getenv ("QUERY_STRING");</pre>	Строка query_string содержит последовательность символов, переданных через форму
---	--

Обработка CGI-скриптом данных формы составляет шаблонную часть скрипта. За ней следует содержательная часть, в которой можно анализировать полученные данные, обращаясь при необходимости к различным таблицам баз данных. При этом, возможно, формируется ответ на запрос, выводимый с помощью функции printf() в HTML-документ, а также, если необходимо, динамически строятся HTML-формы, обеспечивая гибкость в видоизменении интерфейса во время использования.

3.5. ДОСТУП К ДАННЫМ ИЗ CGI-СКРИПТА, НАПИСАННОГО НА ЯЗЫКЕ ESQL/C

Дальнейшая логика работы написанного на языке C CGI-скрипта, вызванного с HTML-страницы и получившего через форму входные данные, отличается средствами, с помощью которых ведется работа с базой данных. Это могут быть:

- встроенный язык ESQL/C, описанный в первой части настоящего учебного пособия;
- некоторый API (Application Program Interface), посредством которого обеспечивается интерфейс к базам данных. Этот интерфейс для языка C поддерживается с помощью наборов функций, специфичных для разных СУБД.

Какого-то общего правила в именовании функций и их аргументов, а также процедур практического использования в API различных СУБД, к сожалению, нет.

Правила работы с базой данных CGI-скрипта, написанного на языке ESQL/C, мало отличаются от тех положений, которые описаны в

разделе 1. Работа с базой данных ведется средствами встроенного языка SQL и главных переменных, через которые передаются и возвращаются данные к серверу баз данных.

В основе взаимодействия CGI-программ и базы данных лежат простые принципы. Необходимо:

- подключиться к серверу баз данных и зарегистрироваться;
- выбрать базу данных, которая должна использоваться;
- отправить запрос SQL на сервер и получить данные;
- отключиться от сервера баз данных.

Остановимся на тех особенностях, которые необходимо учесть, так как CGI-скрипт запускается из среды WWW и является, по сути, публичным ресурсом.

1. Обычная программа, написанная на языке ESQL/C и находящаяся в определенном каталоге, черпает информацию о сервере баз данных, с которым она работает, из соответствующих системных файлов. CGI-скрипт, запускаемый из среды WWW, такой возможности не имеет и должен содержать информацию о сервере баз данных в своем теле. Эта информация о CGI-скрипте задается в системных переменных, определяющих каталог с программным обеспечением сервера, имя сервера, параметры перекодировки и прочие параметры.

```
// Объявление переменных
putenv ("INFORMIXSERVER=fpm");
putenv ("INFORMIXDIR=/usr/Informix");
putenv ("INFORMIXSQLHOSTS=${INFORMIXDIR}/etc/sqlhosts");
putenv ("PATH=${INFORMIXDIR}/bin:${PATH}");
putenv ("ONCONFIG=onconfig");
putenv ("DB_LOCALE=ru_RU.UTF-8");
putenv ("CLIENT_LOCALE=ru_RU.UTF-8");
```

2. Важной особенностью работы с базой данных посредством CGI-скрипта является обеспечение достаточных условий безопасности в отношении данных. Следует отметить, что безопасность – одно из слабых мест CGI-технологии: слишком много уязвимостей имеется при

неосторожном ее использовании. Однако минимальный уровень защиты должен быть соблюден.

Безопасность данных напрямую зависит от тех действий, которые выполняются в приложении. Например, если приложение занимается только выборкой данных, то вероятность возможных повреждений в базе данных будет существенно ниже, чем в случае, если приложение занимается модификацией данных. Однако даже при этом желательно избирательно подходить к вопросу предоставления данных тому или иному пользователю. Многое, конечно, зависит от тех функций, которые выполняет программное приложение. Вопрос этот очень сложный, и можно дать лишь самые общие рекомендации по организации CGI-приложения в целях защиты данных и предоставления полномочий на доступ к базе данных.

Предоставление всем пользователям равных полномочий на доступ к данным

Когда необходимо, чтобы любой пользователь без ограничений имел доступ к данным, проблему целесообразно решить следующим образом.

Предварительно средствами программы dbaccess либо иного программного приложения владелец базы данных SQL-оператором

Grant connect to nobody

предоставляет пользователю операционной системы nobody минимальные права, в том числе в отношении баз данных. Пользователем nobody является, в частности, любой работающий в WWW-среде. Это позволяет при написании CGI-скрипта использовать простейший вариант подключения к серверу баз данных без проверки индивидуальных полномочий пользователя.

Подключение к базе данных
database на сервере onfpm

...

\$Connect to 'database@onfpm'; /* в условиях grant connect to nobody */

Разумеется, что приложение, обеспечивающее любому пользователю беспрепятственный доступ к базе данных, должно гарантировать, что он не выполнит в рамках этого приложения бесконтрольных действий, нарушающих целостность базы данных. После окончания работы с приложением владелец базы данных должен отобрать полномочия у пользователя nobody.

Revoke connect from nobody

Предоставление пользователям индивидуальных полномочий на доступ к данным

Когда необходимо контролировать доступ того или иного пользователя к базе данных или когда разные пользователи обладают разными полномочиями в отношении доступа к данным, следует проверять эти полномочия при подключении к серверу баз данных. Сделать это можно как минимум двумя путями:

- добавить на сервере баз данных учетную запись для пользователя, дать ему требуемые полномочия на те или иные действия над используемыми таблицами посредством SQL-оператора Grant, после чего каждый раз при обращении проверять эти полномочия. При этом, конечно, CGI-приложение должно знать и каким-либо образом хранить все имена, пароли и полномочия пользователей, которым разрешен доступ к данным.

Подключение к базе данных database на сервере onfpm пользователя name1 с паролем qwerty12

```
$char parol[8]= "qwerty12";
```

```
....
```

```
$Connect to 'database@onfpm' user 'name1' using $parol;
```

- самостоятельно создать и поддерживать базу данных пользователей, проверяя их полномочия на выполнение тех или иных действий. Вся ответственность за сохранность и корректное использование данных в этом случае возлагается на разработчика CGI-скрипта. Работа с CGI-скриптом ведется в режиме

Grant connect to nobody

Существует большое количество подходов к решению этой чрезвычайно важной проблемы безопасности, описанных в специальной литературе.

3. Все действия по работе с базой данных, как уже отмечалось, выполняются средствами встроенного ESQL/C, описанными в разделе 1.

4. Результаты обработки данных, полученных из базы данных, а также другие информационные сообщения передаются функцией `printf()` в выходной поток с учетом форматирования HTML-документа.

Приведем пример CGI-скрипта, выводящего количество и перечень поставок поставщика, имя которого введено в HTML-форме:

Введите фамилию поставщика:	<input type="text" value="Smith"/>
<input type="button" value="Послать запрос"/>	<input type="button" value="Отменить запрос"/>

```
.....  
char String[50];  
int j, i;  
$int kolich, kol;  
$char parol[8]= "qwerty12";  
$char post[5], vspom[20], det[6], izd[6];  
.....  
printf("Content-Type: text/html\n\n");  
printf("<HTML><BODY>");  
i=0;  
while ((Value=getc(stdin))!=EOF) { String[i]=Value; i+=1; };  
String[i]=0;  
i=0;
```

Объявление переменных окружения СУБД

Получение строки символов, выданной методом POST

Окончание фрагмента программы

```

while (String[i] != '=') i+=1;
    i+=1; j=0;
while (String[i] != '&') { vspom[j]=String[i]; i+=1; j+=1; };
$Begin Work;
    $Connect to 'database@onfpm'; /* with grant connect to
nobody */
    Error_Messages();
    $select n_post into $post from s where name=$vspom;
    Error_Messages();
    $declare cust cursor for
        select n_det,n_izd,kol from spj where n_post=$post;
    Error_Messages();
$open cust;
    Error_Messages();
    while (sqlca.sqlcode != 100)
        {
            $fetch cust into $det, $izd, $kol;
            if (sqlca.sqlcode == 0)
                printf("<BR>%s | %s | %d<BR>", det, izd, kol);}
$close cust;
$select sum(kol) into $kolich from spj
        where n_post=$post;
printf("<BR>Объем - %d<BR>", kolich);
$Disconnect Current;
$Commit Work;
printf("</BODY></HTML>");
}

```

Выделение первого параметра формы (фамилия поставщика)

Соединение с базой данных

Анализ SQL-ошибок

Определение номера поставщика

Работа с курсором

Выбор по курсору

Вывод результатов выборки в HTML-форму

Определение объема поставок

Вывод данных о поставках в HTML-форму

Отключиться от базы данных, завершить транзакцию и закончить HTML-документ

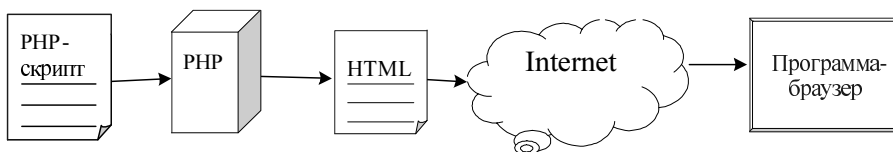
5. Дополнительную гибкость приложениям придает динамическое формирование HTML-форм в CGI-скриптах, когда CGI-программа анализирует введенные пользователем данные и порождает новую HTML-форму для ввода дополнительной информации. Делается это посредством той же функции `printf()`, выводящей в выходной поток атрибуты тэга `<FORM>` в соответствии с ее синтаксисом и с учетом особенностей форматирования HTML-документа. Особой сложности это не вызывает, однако требует определенной тщательности, поскольку в элементах формы могут встречаться символы `<`, `«`, `>`, которым должен предшествовать символ обратной наклонной черты.

Подведем некоторые итоги. В настоящее время технология CGI уже не занимает доминирующую позицию в Интернете. К недостаткам CGI можно отнести трудности в решении проблемы безопасности. Слишком много изъянов в защите может появиться при неосторожном ее использовании. Низка и производительность CGI. Если программа открывает соединение с базой данных, то для нового экземпляра CGI-соединение должно быть открыто заново, что резко ограничивает число CGI-запросов. Однако эти недостатки компенсируются тем, что:

- CGI-интерфейс прост концептуально;
- CGI является открытым стандартом, поддерживаемым большинством серверов независимо от аппаратной части и операционных систем;
- CGI-программы легко писать, используя различные языки программирования;
- существует множество свободно распространяемых сценариев.

4. ИСПОЛЬЗОВАНИЕ ЯЗЫКА PHP ДЛЯ ДОСТУПА К БАЗАМ ДАННЫХ

PHP – это действующий на стороне сервера встраиваемый в HTML межплатформенный язык сценариев, имеющий синтаксис, близкий к языку C. Язык PHP дает возможность вставлять в файлы HTML инструкции языка PHP для создания динамического содержания. Эти инструкции обрабатывает препроцессор-интерпретатор PHP и заменяет их тем содержимым, которое производит этот код. PHP-программа может целиком состоять из конструкций языка PHP, а может быть смесью конструкций языков PHP и HTML. Стандартное расширение файла с PHP-программой – php.



Одним из распространенных применений PHP является работа с базами данных. Для целого ряда баз данных PHP имеет собственную поддержку, а другие доступны через ODBC-функции PHP. При вызове PHP-программы URL-адрес должен содержать номер порта, через который работает PHP:

`html://fpm.ami.nstu.ru:81/~pmxxyy/t1.php`

4.1. СИНТАКСИС ЯЗЫКА PHP

В именах переменных PHP верхний и нижний регистры различаются, следовательно, `$variable` и `$Variable` – две разные переменные. Однако в именах функций PHP регистр не играет роли. Операторы PHP заканчиваются точкой с запятой.

В PHP присутствуют три типа комментариев:

- /* комментарии */
- // комментарии
- # комментарии

Встраивание PHP в HTML. При встраивании кода PHP в документ HTML для этого файла нужно использовать одно из следующих расширений: .phtml, .PHP, .PHP3, чтобы сервер знал о необходимости послать файл на обработку PHP.

Код PHP встраивается в обычную страницу HTML. Существует несколько способов встраивания кода PHP в документ HTML.

1. Поместить код между тэгами `<? и ?>`:

```
<? echo "Hello World"; ?>
```

2. Поместить код между тэгами `<?PHP и ?>`:

```
<?PHP echo "Hello World"; ?>
```

3. Использовать тэги `<script>`:

```
<script language="PHP"> echo "Hello World"; </script>
```

Можно встраивать сразу несколько команд, разделяя их точкой с запятой:

```
<?
    echo "первый оператор";
    echo "второй оператор";
?>
```

Допустимо в любой момент переключаться с HTML на PHP и обратно.

Следующий пример иллюстрирует процедуру вывода десяти строк с переключением между контекстами HTML и PHP:

```
<? for ($i=0; $i<10; $i++)
    { ?>
        <b> Одна из многих строк </b>
    <? }
?>
```

Включение файлов. Важной функцией PHP является включение файлов, в которых могут содержаться дополнительные тэги PHP. Для этого используется ключевое слово `include`, включающее файл `page.inc`.

```
<? $body = "Тело документа" ?>
<HTML><BODY>
    <? echo $body; ?>
</BODY></HTML>
```

в PHP-программу, как показано ниже.

```
<?
    // действия ...
    include "page.inc"
    // действия ...
?>
```

Переменные. В PHP-программе имена всех переменных начинаются с символа `$`. Переменная автоматически объявляется при первом присвоении ей значений. Переменные PHP не типизированы, им можно присваивать значения любого типа данных.

```
<?php
    $message = "Hello, World!";
    echo $message;
?>
```

Типы данных. PHP поддерживает три простых типа данных: целые числа, числа с плавающей запятой и строки, а также два составных типа данных: массивы и объекты. В зависимости от контекста выполняется приведение типов.

Строки. Строка – последовательность символов, заключенная в одиночные или двойные кавычки, при этом строка, заключенная в двойные кавычки, означает, что встречающиеся в этой строке переменные будут заменены их значениями:

```
<?php
    $name = 'Susannah';
    $greeting_1 = "Hello, $name!";
    $greeting_2 = 'Hello, $name!';
    echo "$greeting_1\n";      ──────────> Hello, Susannah!
    echo "$greeting_2\n";      ──────────> Hello, $name!
?>
```

Массивы. Массив представляет собой тип данных, в котором может содержаться множество значений, индексируемых с помощью чисел или строк. Например:

```
$array[0] = "string number one";
$array[1] = "string number two";

или

$Month["January"] = 1;
$Month["December"] = 12;
```

Массив, значения которого индексируются при помощи строковых значений, называют ассоциативным.

Для добавления элемента в конец массива можно использовать сокращенную запись, в которой индекс не указывается.

```
$array[] = "string to end"
```


Аналогично организуется работа с многомерными массивами:

```
$people["David"]["shirt"] = "blue";  
$people["David"]["car"] = "minivan";  
$people["Adam"]["shirt"] = "white";  
$people["Adam"]["car"] = "sedan";
```

Создать массив можно путем вызова функции **array()**:

```
$array = array("string number one", "string number two");
```

Встроенная функция **count()** выдает число элементов в массиве:

```
$fruit = array('banana','papaya');  
print count($fruit); —————→ 2
```

Объекты. Объект – это составной тип данных, способный содержать любое количество переменных и функций. В PHP существует лишь самая базовая поддержка объектов, ее цель – облегчить инкапсуляцию структур данных и функций, чтобы упаковать их в классы для многократного использования.

```
class Sample {  
    var $str = "string";
```

```
class Sample {  
    var $str = "string";  
    function ChageStr($newstr) {    $str = $newstr; }  
}
```

```
$class = new Sample;
echo $class->str;
$class->ChangeStr("new string");
echo $class->str;
```

Создание объекта класса
Sample

Управляющие структуры. Управляющие конструкции языка PHP напоминают конструкции языка C.

1. Оператор If – обычный условный оператор.

```
if (выражение)
{ операторы }
elseif(выражение)
{ операторы }
else
{ операторы }
endif;
```

```
if (выражение):
    операторы
elseif (выражение):
    операторы
else:
    операторы
endif;
```

2. Оператор Switch можно использовать вместо серии операторов if.

```
switch (выражение) {
case выражение:
    операторы
    break;
default:
    операторы
    break;
}
```

```
switch (выражение)
case выражение:
    операторы
    break;
default:
    операторы
    break;
endswitch;
```

3. Оператор While – это программная конструкция цикла, в котором выполнение некоторого кода повторяется, пока истинно некоторое выражение.

```
while (выражение)
{ операторы }
```

```
while (выражение)
    операторы
endwhile;
```

4. Оператор do/while – аналогичен while, но выражение, представляющее условие выполнения цикла, вычисляется в конце каждой итерации, а не перед ее началом.

```
do
    { операторы }
while (выражение)
```

5. Цикл for:

```
for (начальное_выражение; условное_выражение; итеративное_выражение)
    { операторы }
for (начальное_выражение; условное_выражение; итеративное_выражение):
    операторы
endfor;
```

Функции. Функция – это именованная последовательность операторов, которая при необходимости может принимать параметры и возвращать значение. PHP предоставляет большой набор встроенных функций. PHP также поддерживает функции, определяемые пользователем. Для определения функции используется ключевое слово `function`.

```
function max($a, $b)
{
    if ($a < $b) { return $b; }
    else { return $a; }
}
```

Имена определяемых пользователем функций не должны совпадать с именами встроенных функций PHP и ранее объявленными именами переменных.

При определении функций есть возможность указывать для параметров значения по умолчанию.

```
function max ($a =1, $b =2)
{
    if ($a < $b) { return $b; }
    else { return $a; }
}
```

Существуют два способа передачи аргументов функции: по значению и по ссылке. При передаче первым способом значение аргумента присваивается соответствующему параметру, определенному в функции. Изменение этого параметра внутри тела функции не влияет на аргумент, переданный функции. При передаче аргумента по ссылке все изменения параметра внутри функции отразятся и на значении аргумента.

Для указания, что функция принимает аргумент по ссылке при декларации функции, перед значением аргумента ставится символ &.

```
function mult($a, &$b) {
    $b = $b * $a;
    return $a = $a*$b;
}
```

Язык PHP имеет целый набор встроенных функций, которые выполняют стандартные действия, позволяющие, например:

- **print (string);** – направить вывод из сценария в браузер;
- **isset(variable);** – определить, установлено ли значение переменной. Например, вызов функции **isset()** со значением `$submit` позволяет определить, послал ли пользователь данные из формы или нет.

4.2. РАБОТА С ФОРМАМИ

Одно из преимуществ PHP – способность автоматически передавать значения переменных из HTML-форм в переменные PHP. PHP автоматически генерирует набор переменных, имена которых совпадают с именами объектов в HTML-форме и содержат значения данных объектов.

Например, после нажатия кнопки «SUBMIT» на странице, содержащей HTML-форму

```
<FORM ACTION="script.php">
    <INPUT TYPE=TEXT NAME="name" VALUE="Smith">
    <INPUT TYPE=SUBMIT>
</FORM>
```

во время выполнения программы script.php, переменная \$name будет иметь значение «Smith», которое можно использовать в соответствующем контексте, например,

```
if ($name == "Smith") { echo "Please check your email."; }
```

Разница в простоте написания данного PHP-скрипта по сравнению со скриптом, написанным на языке C, требующем разбора последовательности

```
имя=значение&имя1=значение1&...&имяN=значениеN
```

очевидна.

В завершение приведем пример обработки данных, введенных посредством формы, включающий текст HTML-скрипта формы и ее внешний вид:

```
<HTML>
<HEAD>
    <title>Пример обработки формы PHP</title>
</HEAD>
<body>
<form ACTION="t3.php">Имя поставщика: <INPUT
TYPE=TEXT NAME="name"><BR>
<br>Поставляемая деталь: <INPUT TYPE=RADIO
NAME="n_det" VALUE="P1">P1
<INPUT TYPE=RADIO NAME="n_det" VALUE="P2"> P2
<INPUT TYPE=RADIO NAME="n_det" VALUE="P3"> P3
```

Окончание фрагмента программы

```
<br>Изделия, для которых выполнялись поставки:
<INPUT TYPE=CHECKBOX NAME="times[]" VALUE="J1">J1
<INPUT TYPE=CHECKBOX NAME="times[]" VALUE="J2">J2
<INPUT TYPE=CHECKBOX NAME="times[]" VALUE="J3">J3
<INPUT TYPE=CHECKBOX NAME="times[]" VALUE="J4">J4
<br><INPUT type="submit" name="submit" value="OK">
<INPUT type="reset">
</form>
</body>
</HTML>
```

Результат выполнения
HTML-документа

Имя поставщика:

Поставляемая деталь: ☐ P1 ☒ P2 ☐ P3

Изделия, для которых выполнялись поставки: ☒ J1 ☐ J2 ☒ J3 ☒ J4

После ввода данных в форму, как показано выше, запускается PHP-программа, выполняющая обработку параметров формы:

```
<?php
if ($name != "")
{ $string="Поставщик $name поставил деталь $n_det для изделий";
echo "$string"; }
?>
<br><?php if ($times[0] != "") echo "$times[0]";?>
<br><?php if ($times[0] != "") echo "$times[1]";?>
<br><?php if ($times[0] != "") echo "$times[2]";?>
<br><?php if ($times[0] != "") echo "$times[3]";?>
<br><?php $favorite_times = count($times);
echo "Общее число поставок: $favorite_times»;
?>
```


Окончание таблицы

<p>2. resource Pg_query (id_соединения, строка_запроса)</p> <p>Входные параметры:</p> <p>Возвращаемое значение</p>	<p>Выполнение запроса к базе</p> <p>d_соединения – идентификатор соединения; строка_запроса – строка SQL-запроса; Результат в виде ресурса или FALSE</p>
<p>3. array Pg_fetch_row(resource результат_запроса, int номер_строки)</p> <p>array Pg_fetch_assoc (resource результат_ запроса, int номер_строки)</p> <p>Входные параметры:</p> <p>Возвращаемое значение</p>	<p>Получить строку запроса как нумерованный массив</p> <p>Получить строку запроса как ассоциативный массив</p> <p>результат_запроса – идентификатор результата, возвращенный функцией Pg_query() (только для запросов типа select); номер_строки – целое число</p> <p>Строка таблицы базы данных, возвращаемая как массив</p>
<p>4. int Pg_affected_rows (resource result_id)</p> <p>Входные параметры:</p> <p>Возвращаемое значение</p>	<p>Получение числа столбцов, обработанных запросом</p> <p>result_id – результат, возвращенный функцией Pg_query()</p> <p>Возвращается число столбцов, обработанных запросом, ассоциированных с result_id. Для вставок, обновлений и удалений – это реальное количество обработанных столбцов. Для выборок – ожидаемое количество</p>
<p>5. bool pg_free_result (resource result_id)</p> <p>Входные параметры:</p> <p>Возвращаемое значение</p>	<p>Освобождение ресурсов запроса</p> <p>result_id – результат, возвращенный функцией Pg_query()</p> <p>Освобождает ресурсы, занятые запросом с идентификатором результата result_id. Возвращает 0 в случае ошибки</p>
<p>6. bool pg_close (resource link_identifier)</p> <p>Входные параметры:</p> <p>Возвращаемое значение</p>	<p>Закрытие соединения с PostgreSQL</p> <p>link_id – идентификатор соединения;</p> <p>1 в случае успешного закрытия соединения с PostgreSQL и 0 в случае ошибки</p>

Функции взаимосвязи с СУБД INFORMIX

<p>1. int ifx_connect (string database, string user, string password) Входные параметры:</p> <p>Возвращаемое значение</p>	<p>Создание соединения с сервером Informix</p> <p>database – имя базы данных; user – имя пользователя; password – пароль</p> <p>идентификатор соединения, если соединение прошло успешно, и равен 0 в противном случае</p>
<p>2. int ifx_query (string query, int link_id, int cursor_type) Входные параметры:</p> <p>Возвращаемое значение</p>	<p>Выполнение запроса к базе</p> <p>query – строка SQL-запроса; link_id – идентификатор соединения; cursor type – тип курсора</p> <p>значение 1 или 0 в зависимости от успеха выполнения операции</p>
<p>3. array ifx_fetch_row (int result_id, mixed [position]) Входные параметры:</p> <p>Возвращаемое значение</p>	<p>Получение строки запроса в виде массива</p> <p>result_id – идентификатор результата, возвращенный функцией ifx_query() (только для запросов типа select); [position] – параметр из списка: next, previous, current, first, last или номер строки</p> <p>строка таблицы базы данных, возвращаемая как массив</p>
<p>4. string current (array row) Входные параметры:</p> <p>Возвращаемое значение</p>	<p>Получение очередного поля из строки таблицы базы данных</p> <p>array row – строка таблицы базы данных, возвращенная функцией ifx_fetch_row()</p> <p>очередное поле строки таблицы</p>
<p>5. string next (array row) Входные параметры:</p> <p>Возвращаемое значение</p>	<p>Получение следующего поля из строки таблицы базы данных</p> <p>array row – строка таблицы базы данных, возвращенная функцией ifx_fetch_row()</p> <p>следующее поле строки таблицы</p>

Окончание таблицы

6. int reset(array\$row) Входные параметры: Возвращаемое значение	Переход в начало строки array row – строка таблицы базы данных, возвращенная функцией ifx_fetch_row() нулевая позиция строки результата
7. string key(array\$row) Входные параметры: Возвращаемое значение	Получение имени столбца array row – строка таблицы базы данных, возвращенная функцией ifx_fetch_row() имя очередного столбца из поля строки результата
8. int ifx_affected_rows int result_id) Входные параметры: Возвращаемое значение	Получение числа столбцов, обработанных запросом result_id – результат, возвращенный функцией ifx_query() число столбцов, обработанных запросом, ассоциированным с result_id. Для вставок, обновлений и удалений – это реальное количество обработанных столбцов, для выборок – ожидаемое количество
9. int ifx_free_result (int result_id) Входные параметры: Возвращаемое значение	Освобождение ресурсов запроса result_id – результат, возвращенный функцией ifx_query() 1 в случае успешного освобождения ресурсов, занятых запросом, и 0 в случае ошибки
10. int ifx_close (int [link_identifier]) Входные параметры: Возвращаемое значение	Закрытие соединения с Informix link_id – идентификатор соединения 1 в случае успешного закрытия соединения с Informix и 0 в случае ошибки

Общая схема написания РНР-программы, выполняющей взаимодействие с базой данных, мало отличается от структуры CGI-скрипта, написанного любыми другими средствами, и включает:

- подключение к серверу баз данных и регистрацию;
- выбор базы данных, которая будет использоваться;
- посылку SQL-запроса на сервер и получение данных;
- отсоединение от сервера баз данных.

При этом остаются актуальными все замечания, сделанные в разделе 3 относительно установки переменных окружения и обес-

печения мер безопасности при работе с базой данных. Следующий пример демонстрирует использование средств PHP для работы с базой данных.

```

<?
putenv ("INFORMIXSERVER=onfpm");
putenv ("INFORMIXDIR=/usr/informix");
putenv ("INFORMIXSQLHOST=/usr/informix/etc/sqlhosts");
putenv ("INFORMIXSERVER=onfpm");
putenv ("DB_LOCALE=ru_ru.KOI-8");
putenv ("CLIENT_LOCALE=ru_ru.KOI-8");
$res=ifx_connect("database@onfpm","name1","querty12");
$rid=ifx_query("select * from spj", $res);
if (!$rid)
{
    printf("Error: %s",ifx_error()); die();
}
$rowcount = ifx_affected_rows($rid);
echo ("result: <b>$rowcount</b> rows<br>");
$row = ifx_fetch_row ($rid, "NEXT");
while (is_array($row))
{
    // пока переменная $row не пуста
    $n_post=current($row);
    $name=next($row);
    $reiting=next($row);
    $town=next($row);
    printf("%s %s %s %s", $n_post, $name, $reiting, $town);
    printf("\n<br>");
}
    
```

Объявления переменных окружения Informix

Соединение с сервером баз данных

Выполнение SQL-запроса

Проверка корректности выполнения SQL-запроса

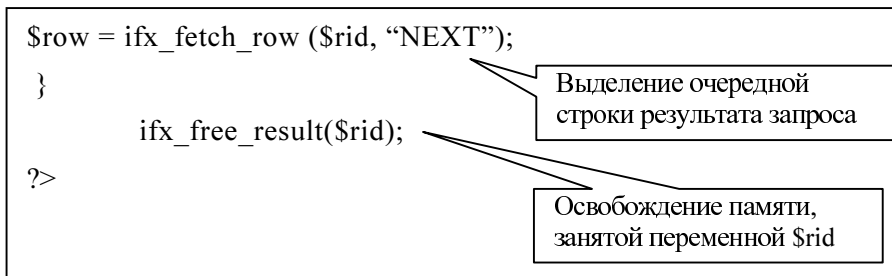
Получение количества строк в результате запроса

Выделение первой строки результата запроса

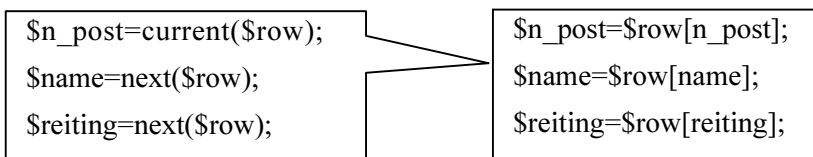
Выделение полей строки результата запроса

Вывод в HTML

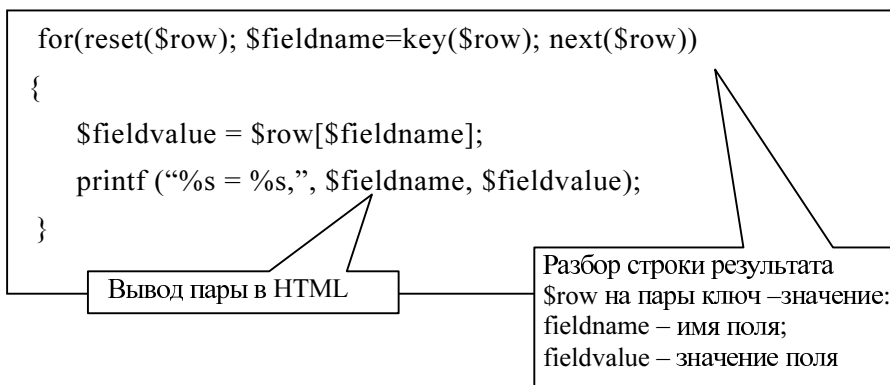
Окончание фрагмента программы



Альтернативой использованию функций **current()**, **next()** служит выбор полей из строки результата по их именам:



Посредством функций можно произвести несложный разбор строки результата \$row на пары имя поля – значение:



Приведенная ниже РНР-программа объединяет действия по вводу данных посредством формы и последующую работу с базой данных:

```

<html>
<head>
<title>Пример использования PHP при работе с базой данных</title>
</head>
<body>
<?php
    if (isset($submit)):
        {echo "Список деталей и изделий:<BR>";
    putenv ("INFORMIXSERVER=onfpm");
    putenv ("INFORMIXDIR=/usr/informix");
    putenv ("INFORMIXSQLHOST=/usr/informix/etc/sqlhosts");
    putenv ("INFORMIXSERVER=onfpm");
    putenv ("DB_LOCALE=ru_ru.KOI-8");
    putenv ("CLIENT_LOCALE=ru_ru.KOI-8");
    $db =ifx_connect("database@onfpm","name1","querty12");
    $sql = "select * from spj where n_post='$x'";
    $result=ifx_query($sql, $db);
    if (!$result) echo "Error";
    else {
        while($row = ifx_fetch_row ($result, "NEXT")){
            $n_det=$row["n_det"];
            $n_izd=$row["n_izd"];
            print("$n_det  $n_izd<BR>\n");
        }
        ifx_free_result($result);
    }
}
?>
<?php

```

Проверка, нажата ли клавиша ввода

Обработка данных, введенных через форму, и выполнение на основе их SQL-запроса

x – параметр, введенный через форму

Проверка корректности запроса

Выборка очередной строки, выделение из нее полей и вывод их в HTML

Окончание фрагмента программы

```
else :?>
```

```
<form>
```

```
<p>Введите ID: <input type="text" name="x">
```

```
<p><input type="submit" name="submit" value="OK">
```

```
<input type="Reset">
```

```
</form>
```

```
<?
```

```
endif;
```

```
?>
```

```
</body>
```

```
</html>
```

Вывод формы и ввод
через нее данных
для формирования
SQL-запроса

Для хранения небольших объемов информации на стороне клиента существует технология Cookies. Хранение информации реализует Web-сервер, а передачу – протокол http.

Для создания и модификации значения cookie можно использовать функцию PHP **Setcookie()**. Эта функция может иметь до шести аргументов в зависимости от того, какие функции Cookie используются и кто будет считывать ее значения.

Простейший способ установки Cookie в виде `setcookie('name', 'bret')` присваивает переменной \$name значение 'bret' для каждой последующей страницы на WWW-сайте, просматриваемой в течение данной сессии (пока пользователь не покинет сайт), что позволяет контролировать значение этой переменной средствами PHP. Этот тип Cookie известен как Cookie-сессия, поскольку значение сохраняется в течение пользовательской сессии.

Если желательно, чтобы значение Cookie запоминалось браузером после того как пользователь закончит сессию, необходимо передать функции **Setcookie()** третий параметр – дату истечения срока действия Cookie. Вспомогательная функция `Mktime()`, параметрами которой

являются час, минута, секунда, месяц, день и год, возвращает число секунд, прошедших с 1 января 1970 г. до указанного момента времени.

Следующие два параметра функции **Setcookie()** позволяют задать путь и имя домена того, кто может прочесть значение вашего Cookie. По умолчанию из соображений безопасности только HTML-страницы, расположенные в том же каталоге или ниже в структуре подкаталогов того сервера, который установил Cookie, могут прочесть его значение.

Последний параметр функции **Setcookie()**, равный 1, требует, чтобы значение Cookie передавалось только на те Web-серверы, которые используют безопасный протокол соединения (SSL – Secure Socket Layer). Для удаления Cookie достаточно передать функции **Setcookie()** единственный параметр – имя Cookie : `setcookie('name')`.

Использование технологии Cookie демонстрируется приведенным ниже Web-приложением, позволяющим посетителям сайта проголосовать за то или иное средство разработки. Структура таблицы базы данных для внесения средств разработки будет выглядеть так:

```
CREATE TABLE names (  
    name  varchar(30) NOT NULL,  
    votes  int(4),  
    PRIMARY KEY (name) );
```

Текстовое поле предназначено для названия, а целочисленное – для подсчета голосов, поданных за указанное средство разработки.

Приложение должно:

- заносить данные голосования;
- обеспечивать минимальную проверку, не позволяя голосовать несколько раз подряд;
- выводить диаграмму, отражающую относительную долю проголосовавших за каждое из средств.

Формировать столбцы диаграммы можно одним из трех способов:

1) изменяя значения атрибутов `height` и `width` тэга ``, добиться масштабирования одного небольшого изображения в соответствии с процентным отношением каждого из средств;

2) используя вложенные таблицы и устанавливая фиксированные значения на высоту и ширину столбцов и используя заливку ячеек таблицы, можно также получить изображение столбцовой диаграммы;

3) используя встроенные функции PHP для формирования GIF-изображений (библиотека GD).

В создании приложения выбран простейший путь – масштабирование изображения.

```
<?php
putenv("INFORMIXDIR=/usr/informix");
putenv("INFORMIXSERVER=onfpm");
putenv("INFORMIXSQLHOSTS=/usr/informix/etc/sqlhosts");
putenv("DB_LOCALE=ru_ru.KOI-8");
putenv("CLIENT_LOCALE=ru_ru.KOI-8");
if ($vote && !$already_voted) SetCookie("already_voted", "1");
?>
<HTML>
<BODY>
<H3>Укажите средство разработки, которое Вы предпочитаете</H3>
<form action = "test.php" method=post>
  Название: <input type="text" name=new_name>
  <input type="submit" value="Проголосовать">
<?
$conn_id = ifx_pconnect("vms@onfpm", "vms", "qwerty12");
$db = "test";
$table = "names";
if (!$conn_id) {
    echo "Ошибка в соединении с базой данных";
    die();
}
```

Объявление внешних переменных, необходимых для работы СУБД Informix

Занесение данных о проведении голосования, если голосование проводится впервые

Установка связи с сервером баз данных

Остановка выполнения скриптов PHP

Продолжение фрагмента программы

```
if ($new_name) {
if (@!ifx_query("insert into $table values('$new_name', 1)",
$conn_id))
    { echo ifx_errormsg()."<br>"; }
}
if ($vote && $already_voted) {
    echo "<p>Вы уже голосовали! Попытка игнорируется!</p>";
} else if ($vote) {
    if (!ifx_query("update $table set votes = votes + 1 where name
= '$vote' ", $conn_id)) {
        echo ifx_errno().": ".ifx_error()."<BR>";
    }
    $result = ifx_query("select sum(votes) as sum from $table",
$conn_id);
    $row = ifx_fetch_row ($result, "NEXT");
    $sum = (int)$row["sum"];
    ifx_free_result($result);
    $result = ifx_query("select * from $table order by votes DESC",
$conn_id);
    echo "<TABLE BORDER = 0><TR><TH>Голосование</TH>";
    echo "<TH>Предложения</TH><TH COLSPAN =
2>Голоса</TH></TR>";
    $row = ifx_fetch_row($result, "NEXT");
    while (is_array($row)) {
```

Если значение не введено,
условие не выполнится

Попытка выполнить запрос
на внесение данных в таблицу

Добавление голоса в его пользу,
если средство разработки уже
существует в базе данных

Подсчет общего числа
голосов в базе данных

Получение значения

Выбор из таблицы всех записей
для построения диаграммы

Окончание фрагмента программы

```
echo "<TR><TD ALIGN=center>";
echo "<INPUT TYPE=radio Name=vote Value=";
echo $row["name"]."></TD><TD
LIGN=right>".$row["name"]."</TD><TD>";
if ($sum && (int)$row["votes"]/$sum) {
    $per = (int)(100*$row["votes"]/$sum);
    echo "<IMG SRC='point.gif' HEIGHT=12 width=$per>$per
%</TD>"; }
echo "</TR>";
$row = ifx_fetch_row($result, "NEXT"); }
echo "</TABLE>";
ifx_free_result($result);
?>
<input type="submit" value="Проголосовать">
</form> </BODY> </HTML>
```

5. ИСПОЛЬЗОВАНИЕ КОМПОНЕНТ ADO C++ Builder ДЛЯ ДОСТУПА К БАЗАМ ДАННЫХ

В данном разделе кратко излагаются основные методы разработки Windows-приложений для работы с базами данных в интегрированной среде визуального объектно-ориентированного проектирования C++Builder с использованием технологии ActiveX Database Objects (ADO).

5.1. ПРОЕКТИРОВАНИЕ ПРИЛОЖЕНИЙ В ИНТЕГРИРОВАННОЙ СРЕДЕ C++Builder

Приложения, разрабатываемые в среде **C++Builder**, строятся на принципах объектной ориентации. Эти программы – не просто последовательность операторов, не жесткий алгоритм, а описание некоторого набора объектов и способов их взаимодействия. Каждый объект, включаемый в проект, представляет собой совокупность свойств и методов, а также событий, на которые он может реагировать. Приложение управляется последовательностью событий, основным, но не единственным источником которых является пользователь.

Основные используемые для создания приложений в C++ Builder объекты – форма и компонент. Форма – основа приложения, ее назначение – быть контейнером для компонентов. Наиболее часто используется оконная форма (TForm). Для приложений, работающих с базами данных, используется модуль данных (TDataModule). В C++Builder имеется большой набор встроенных компонентов. Каждый компонент предназначен для выполнения определенных задач. Номенклатура компонентов очень обширна и позволяет реализовать практически любое приложение. Кроме того, есть возможность разрабатывать собственные компоненты, а также устанавливать компоненты приобретенные. В дальнейшем мы рассмотрим наиболее часто используемые компоненты.

Приложение (проект) в C++Builder включает в себя одну или несколько форм. На форму с помощью мыши переносятся пиктограммы компонентов, имеющиеся в библиотеках C++Builder. При помощи простых манипуляций можно менять размеры и расположение компонентов, причем результаты видны сразу, даже без компиляции программы. При необходимости можно изменить свойства компонента, а также написать обработчики событий. При этом автоматически формируется код программы, описывающий этот компонент. Таким образом, проектирование в C++Builder сводится к размещению компонентов на форме, заданию некоторых их свойств и написанию обработчиков событий.

При проектировании C++Builder автоматически создает для каждой формы три файла:

- файл формы (.dfm) – хранит информацию о внешнем виде формы, ее размерах, местоположении на экране, о расположенных на ней компонентах и т. д.;
- заголовочный файл модуля(.h) хранит объявления классов формы;
- файл реализации модуля (.cpp) хранит код обработчиков событий формы и ее компонентов.

Также автоматически создаются файлы проекта:

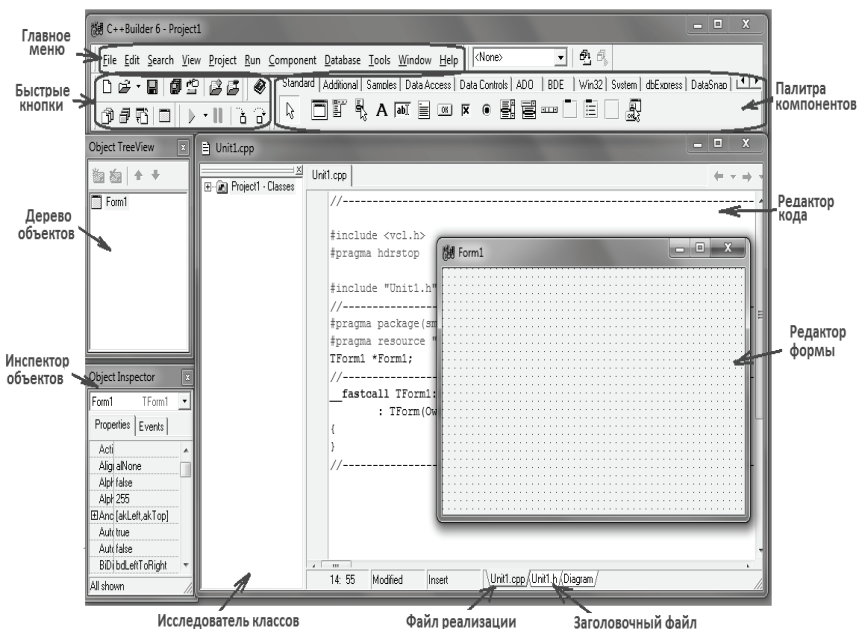
- головной файл проекта (.cpp) хранит главную функцию WinMain, инициализирующую приложение и запускающую его на выполнение;
- файл опций проекта (.bpr) хранит установки опций проекта и указание на то, какие файлы должны компоноваться в проект;
- файл ресурсов проекта (.res) хранит информацию о ресурсах проекта: пиктограммах и т. п.

C++ Builder присваивает всем новым файлам стандартные имена (Form1, Unit1, Project1 и т. д.) Рекомендуется имена для форм, модулей и проектов менять на осмысленные названия при первом сохранении, а для файлов формы и ее модуля выбирать имена одинаковые с точностью до префикса (например FViewQuery и UViewQuery).

5.2. ИНСТРУМЕНТЫ C++Builder

Для работы над проектом C++Builder предоставляет целый набор инструментов.

Главное меню расположено в верхней части окна, назначение разделов и пунктов этого меню станет понятно по мере изложения.



Панель быстрых кнопок включает в себя кнопки для наиболее часто используемых пунктов главного меню.

Палитра компонентов – это витрина библиотеки компонентов. Компоненты сгруппированы по смыслу и назначению. Каждая группа имеет название и расположена на отдельной странице-закладке.

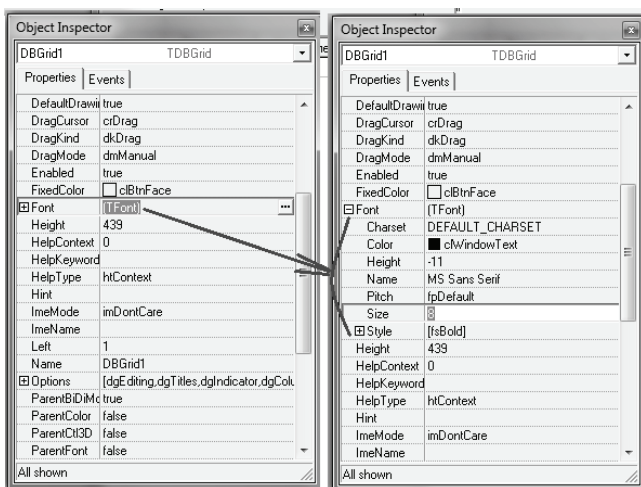
Редактор формы предназначен для работы с формой и ее компонентами в визуальном режиме при помощи мыши. Форма в окне редактора формы покрыта сеткой для удобства размещения на ней компонентов. При выполнении приложения эта сетка не видна. Для добавления компонента в форму нужно найти компонент в палитре, кликнуть на нем мышкой, а затем повторно кликнуть в нужном месте проектируемой формы. Компонент появится на форме, и далее его можно перемещать, менять размеры и другие характеристики.

Редактор кода предназначен для просмотра и изменения кода модуля формы. Нажав на клавишу **F12**, можно быстро перейти из редактора формы в редактор кода и обратно. В редактор кода встроен так называемый **знажок кода**, он подсказывает имена свойств, методов, событий, типы аргументов и многое другое. Например, если

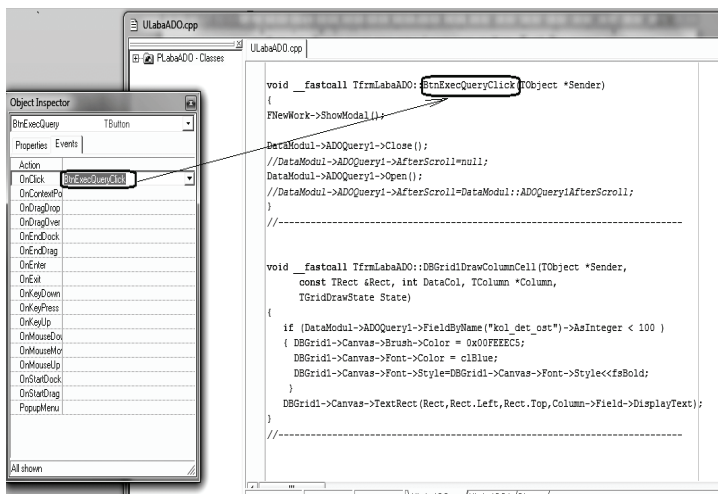
написать в редакторе кода имя компонента, поставить после него символы стрелки(->) и немного задержаться с вводом следующего символа, то появится окно, содержащее список всех свойств, методов и событий класса, к которому принадлежит данный компонент.

Исследователь кода (ClassExplorer) показывает структуру проекта в виде дерева всех типов, классов, свойств, методов, глобальных переменных и глобальных функций, содержащихся в модуле, открытом в редакторе кода.

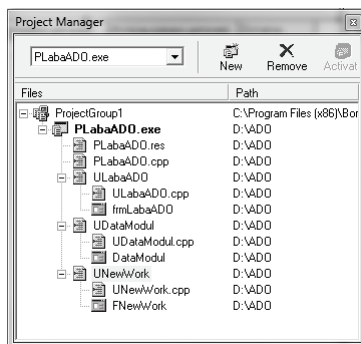
Инспектор объектов (Object Inspector) обеспечивает простой и удобный способ просмотра и изменения свойств объекта и управления событиями объекта. На верхней строке инспектора объектов показывается имя объекта, чьи свойства и события отображаются на расположенных ниже закладках – **Свойства (Properties)** и **События (Events)**. Верхняя строка снабжена выпадающим списком всех компонентов, расположенных на форме, включая саму форму. Чтобы посмотреть свойство какого-либо объекта, нужно выбрать его имя в выпадающем списке или кликнуть на соответствующем компоненте в редакторе формы. Если окно инспектора оказалось скрытым, быстро увидеть его поможет нажатие клавиши **F11**. Рядом с некоторыми свойствами объекта стоит знак «плюс» (например, свойство **Font**). Это значит, что данное свойство является объектом и имеет в свою очередь ряд свойств. Кликнув на «плюс», можно развернуть и посмотреть эти свойства.



Закладка **События (Events)** показывает все события, для которых можно задать собственный обработчик. Двойной клик напротив нужного события позволяет быстро перейти к коду обработчика в тексте модуля формы. Имена функциям-обработчикам событий присваиваются автоматически, менять их не рекомендуется.

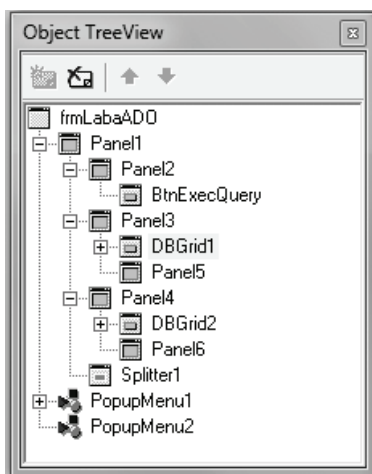


Менеджер проекта (Project Manager) показывает состав форм и модулей приложения и позволяет осуществлять навигацию между ними. Открыть менеджер проекта можно, нажав **Ctrl-Alt-F11** или выбрав в главном меню **View|Project Manager**. В окне менеджера проекта состав проекта отображается в виде иерархического дерева. Для каждого файла указывается, где он расположен на компьютере.



Двойной клик на имени модуля открывает его в окне редактора кода. Кроме того, менеджер проекта позволяет добавить в проект новую или ранее разработанную форму (кнопка New) или удалить ненужную форму из проекта (кнопка Remove).

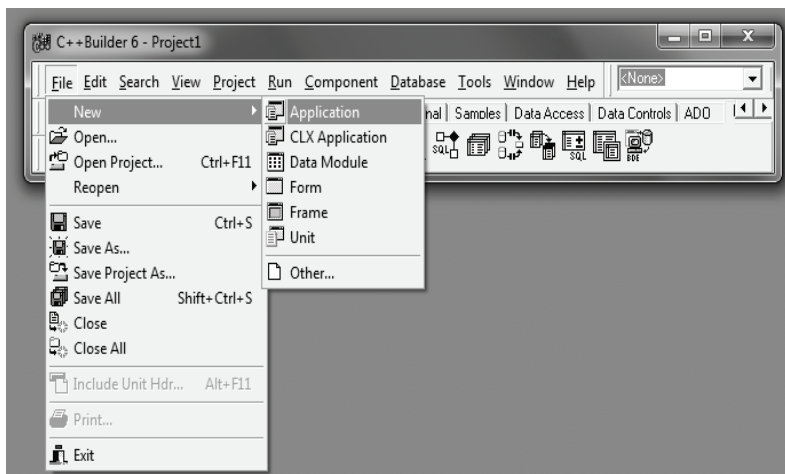
Дерево объектов (Object TreeView) – еще один полезный инструмент. Часто компоненты располагают не непосредственно на форме, а на панелях, зрительно объединяя их в группы (например, по назначению). При этом образуется определенная иерархия вложенности компонентов, которую и показывает дерево объектов. Открыть окно дерева объектов можно, выбрав в главном меню **View|Object TreeView**.



5.3. РАБОТА С ПРОЕКТОМ

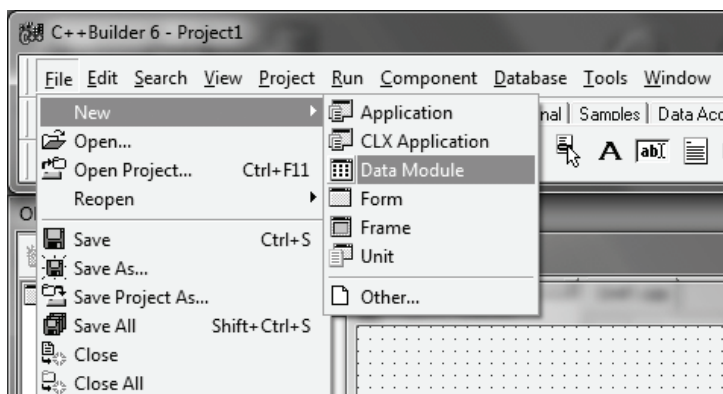
Создание нового проекта

Как правило, при запуске C++Builder автоматически создается новый проект Project1 с единственной формой Form1. Если по какой-либо причине этого не произошло или проект был ранее закрыт, для создания нового проекта необходимо зайти в главное меню **File|New** и в открывшемся подменю выбрать **Application**.



Добавление в проект новой формы

Добавление новой формы в проект выполняется командой главного меню **File|New**. Далее в открывшемся подменю нужно выбрать либо **Form** (добавление новой оконной формы), либо **DataModule** (добавление специального модуля данных). Модуль данных – особый вид формы, используемый в приложениях, работающих с базой данных. Модуль данных – невидимая форма, при разработке проекта он выглядит как обычная форма, куда помещаются компоненты для связи с базой данных. При выполнении приложения модуль данных невидим.



Сохранение проекта

Для сохранения проекта в главном меню есть пункты **File|SaveAll** и **File|SaveProjectAs**.

При выборе **File|SaveAll** сначала сохраняются все файлы форм и модулей, входящих в проект, затем файлы самого проекта.

При выборе **File|SaveProjectAs** сохраняются только файлы самого проекта. Первый раз, перед тем как сохранить проект, рекомендуется сохранить его формы и модули, желательно под осмысленными именами в отдельной папке.

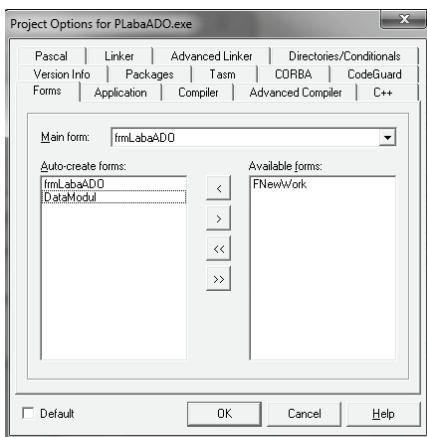
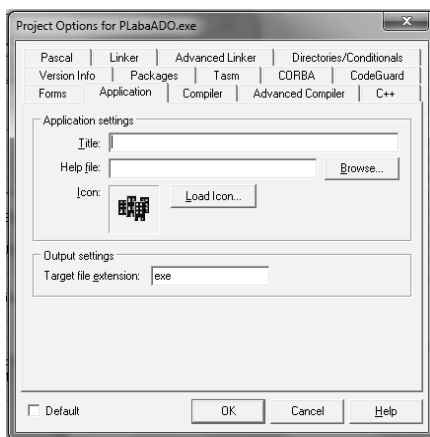
Открытие сохраненного проекта

Для открытия ранее сохраненного проекта используются пункты меню **File|OpenProject** и **File|Reopen**.

Опции проекта

Для доступа к окну опций проекта нужно выбрать пункт главного меню **Project|Options**.

На закладке **Application** можно ввести название приложения и загрузить для него иконку.



На закладке **Forms** в верхнем выпадающем списке можно выбрать главную форму среди имеющихся в проекте. Обычно сколько-нибудь сложное приложение состоит из нескольких форм. По умолчанию все

формы создаются автоматически при запуске приложения и первая включенная в проект форма считается главной. Главной форме передается управление при запуске приложения. При закрытии главной формы выполнение приложения завершается. Главной в приложении может быть вовсе не первая включенная в проект форма, и не всегда нужно все формы создавать сразу в момент запуска. Пользуясь двумя нижними списками, можно установить, какие формы должны создаваться автоматически, а какие – нет.

Закладка Directories/Conditionals позволяет установить директории для размещения/чтения различных типов файлов проекта.

Компиляция, выполнение, отладка

Компиляция модулей проекта, компоновка, подготовка исполняемого файла с расширением .exe и последующее его выполнение осуществляются командой главного меню **Project| Run**, или соответствующей быстрой кнопкой (зеленая стрелка), или нажатием клавиши **F9**. Однако данные процедуры можно выполнять по отдельности. Компиляцию отдельного модуля программы можно выполнить с помощью команды главного меню **Project|Compile**. Команда главного меню **Project|Make** позволяет выполнить компиляцию только тех модулей, которые были отредактированы, после чего выполняет компоновку проекта с созданием исполняемого модуля. Команда **Project|Build** компилирует все модули независимо от того, редактировались они или нет, а затем компоует проект и готовит исполняемый файл.

На этапе компиляции происходит автоматический поиск синтаксических ошибок, неправильного объявления или использования переменных и т. п. При обнаружении подобных ошибок в нижней части редактора кода будет выведено соответствующее сообщение об ошибке или предупреждение. Двойной клик на сообщении позволяет перейти в редакторе кода к строке, содержащей ошибку.

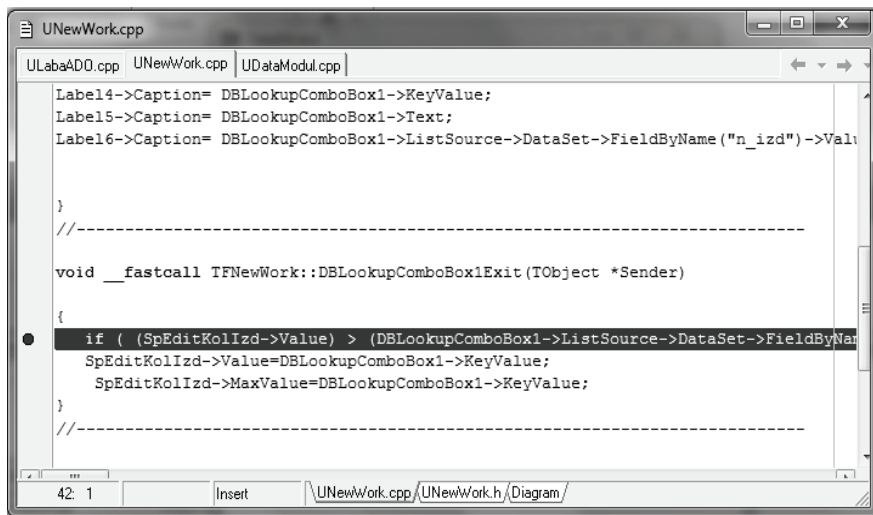
C++Builder предоставляет большой набор средств отладки кода, наиболее часто используемые – пошаговое выполнение и отслеживание состояния переменных.

Пошаговое выполнение программы может использоваться в разных режимах:

- выполнение без захода в вызываемые функции и подпрограммы – по нажатию быстрой кнопки **Step Over** или клавиши **F8**;
- выполнение с заходом в вызываемые функции и подпрограммы – по нажатию быстрой кнопки **Trace Into** или клавиши **F7**;

- переход к следующей исполняемой строке программы – по нажатию быстрой кнопки **Trace to Next** или клавиш **Shif+F7**;
- выполнение до места расположения курсора – по команде **Run to Cursor** или нажатию клавиши **F4**;
- до выхода из выполняемой функции остановится на операторе, следующем за этой функцией, – по команде **Run Until Return** или нажатию клавиш **Shif+F8**.

При необходимости в программном коде можно поставить точки останова, достаточно кликнуть напротив нужной строки на сером поле с левой стороны окна редактора кода.



При отладочном останове программы доступен мастер оценки выражений. Если подвести курсор к имени переменной или свойству объекта, появится текст с текущим значением переменной или свойства.

5.4. ФОРМА, ЕЕ ОСНОВНЫЕ СВОЙСТВА, МЕТОДЫ, СОБЫТИЯ

Форма является основным интерфейсным элементом в C++Builder. С точки зрения Windows форма представляет собой окно. С точки зрения C++Builder форма – это визуальный компонент, играющий роль

контейнера, который содержит другие компоненты, определяющие функциональность приложения.

У формы достаточно много свойств. Первоначально C++Builder устанавливает свойства формы в значения по умолчанию. Изменить свойства формы можно в процессе проектирования (в Инспекторе объектов) или во время выполнения приложения, в последнем случае в исходный текст программы необходимо внести соответствующие операторы.

Основные свойства формы:

- **Name** – имя формы (по умолчанию Form1, Form2,...);
- **Caption** – название окна, отображающееся в строке заголовка;
- **Width** – ширина формы в пикселях;
- **Height** – высота формы в пикселях;
- **Left** – координата формы относительно левой стороны экрана;
- **Top** – координата формы относительно верхней стороны экрана;
- **Position** – стартовая позиция окна;
- **Icon** – иконка в строке заголовка;
- **Color** – цвет фона формы;
- **BorderStyle** – тип границы, обрамляющей форму.

Свойство **BorderStyle** имеет значения:

- bsSizeable – изменяемая в размерах рамка обычного окна;
- bsDialog – неизменяемая в размерах рамка, свойственная окнам диалога;
- bsSingle – неизменяемая в размерах рамка обычного окна;
- bsNone – окно без рамки и заголовка;
- bsToolWindow – рамка аналогична bsSingle, но окно имеет уменьшенный заголовок (целесообразно использовать для служебных окон);
- bsSizeToolWin – рамка аналогична bsSizeable, но с уменьшенным заголовком.

События формы:

- **onCreate** – происходит при создании формы, обработчик этого события может установить начальные значения для свойств формы и ее компонентов, открывать нужные запросы и т. п.;

- **onShow** – происходит непосредственно перед тем, как форма становится видимой;
- **onActivate** – происходит, когда пользователь переключается на форму, т. е. форма получает активность в результате, например, щелчка мыши на ней;
- **onCloseQuery** – происходит при попытке закрыть форму. Попытка может исходить от пользователя, который нажал на рамке формы кнопку «Закрыть», или от программы, которая вызвала у формы метод **Close**. В обработчик события **OnCloseQuery** передается по ссылке булевский параметр **CanClose**, разрешающий или запрещающий действительное закрытие формы;
- **onClose** – происходит после события **OnCloseQuery**, непосредственно перед закрытием формы.

Методы формы

Работая с несколькими формами, следует принимать во внимание, что после загрузки приложения отображается только одна главная форма, остальные формы, даже если они создаются автоматически вслед за главной формой, ждут, пока их вызовут. Форму можно вызвать на исполнение двумя разными способами:




- с помощью метода **Show** для немодального выполнения вместе с остальными формами (режим параллельной работы);
- с помощью метода **ShowModal** для модального выполнения отдельно от остальных форм (режим последовательной работы). Если форма открыта как модальная, до тех пор, пока она не будет закрыта, ни одна другая форма приложения не сможет стать активной.

Форма может быть закрыта либо пользователем (нажатием кнопки «Закрыть» на рамке формы), либо программно с помощью метода **Close**.

В случае, если автоматически создается только главная форма, для всех остальных форм перед их отображением методом **Show** нужно их создавать методом **Create**. А после того как они станут не нужны, удалять методом **OnDestroy**.

5.5. ОБЗОР ПОПУЛЯРНЫХ КОМПОНЕНТОВ




Ниже в таблице приведен перечень наиболее популярных компонентов.

Пикто- грамма	Компо- нент	Страница	Описание
	Label (метка)	Standard	Отображает надпись, которую пользователь не сможет редактировать. У компонента можно менять цвет (свойство Color), шрифт (свойство Font), текст надписи (свойство Caption), видимость (свойство Visible)
	Edit (поле ввода)	Standard	Отображает область редактируемого ввода одиночной строки. Основное свойство – Text – текст редактируемой строки. Даже если вводятся числа, тип свойства Text остается символьным и при использовании в вычислениях его нужно приводить к числовому типу (например, функцией StrToInt). Свойства Color, Font, Visible такие же, как у Label
	MaskEdit (поле маски- рованного ввода)	Additional	Используется для форматирования вводимых данных или ввода по шаблону. Основные свойства: Text – введенный текст EditMask – маска ввода/форматирования EditText – отформатированный текст
	Panel (панель)	Standard	Компонент-контейнер для группировки других компонентов. Свойства Caption Color, Font, Visible аналогичны Label. Свойство Align – выравнивание. По умолчанию это свойство равно alNone, т. е. никакого выравнивания нет. Если задать это свойство равным alTop, панель сместится к верхней границе компонента-контейнера, ее высота останется прежней, а ширина будет равна ширине компонента-контейнера. То же самое произойдет при значении alBottom, только панель сместится к нижней границе. При значениях alLeft и alRight панель сместится к левой или правой границе контейнера соответственно, прежней останется ширина панели, а высота станет равной высоте контейнера. При этом при изменении размеров контейнера размеры вложенной панели будут меняться автоматически

Продолжение таблицы

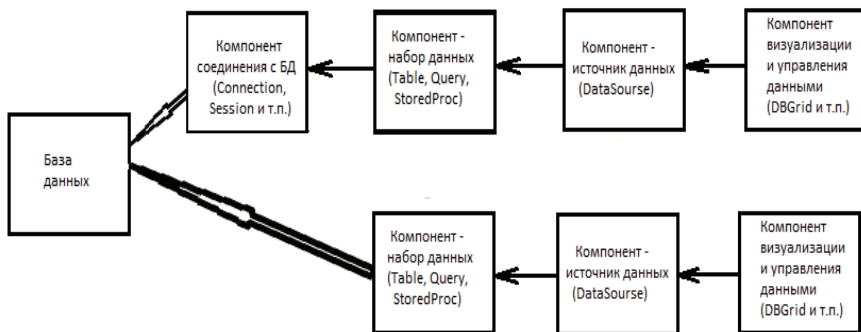
Пикто- грамма	Компонент	Страница	Описание
	Splitter	Additional	Предназначен для ручного (с помощью мыши) управления размерами контейнеров Panel, GroupBox или подобных им во время выполнения программы. Визуально он представляет собой небольшую вертикальную или горизонтальную полосу, располагающуюся между двумя соседними контейнерами или на «свободной» стороне одного из них. Основное свойство – Align аналогично Panel
	Button (кнопка)	Standard	Кнопка с надписью. Основное свойство – Caption – надпись на кнопке. Основное событие – OnClick – клик мышкой на кнопке
	PopUpMenu (контекст- ное меню)	Standard	Контекстное меню для формы или для другого компонента
	CheckBox (флажок)	Standard	Элемент управления с двумя вариантами выбора – его можно установить или снять. Установленный флажок отмечен галочкой. Основные свойства: Caption – текст, отображаемый справа от флажка Checked – сброшен флажок или установлен
	RadioButton (переключатель)	Standard	Флажок с зависимой фиксацией. Основные свойства те же, что у CheckBox. Особенностью радиокнопок является механизм их переключения. Они группируются автоматически, т.е. при выборе одной из них все остальные, принадлежащие одному контейнеру (например, панели или форме), освобождаются. Для того чтобы в форме можно было использовать несколько независимых групп радиокнопок, используют специальные компоненты RadioGroup

Окончание таблицы

Пикто- грамма	Компонент	Страница	Описание
	ComboBox (выпадаю- щий список)	Standard	Поле ввода с возможностью выбора из выпадающего списка. Основные свойства: Items – содержит элементы списка ItemIndex – номер выбранного элемен- та, нумерация начинается с 0. Если ни один элемент не выбран, то значение будет – 1 Text – введенный в поле текст
	CSpinEdit	Samples	Специальный компонент для редакти- рования целых чисел, комбинация обычного поля ввода и кнопки со стрелками вверх и вниз. Основные свойства: MinValue – минимальное допустимое значение MaxValue – максимальное допустимое значение Increment – величина, на которую из- меняется значение при каждом нажа- тии кнопки Value – текущее значение
	DateTimePicker (календарь)	Win32	Поле ввода с возможностью выбора из всплывающего календаря. Основное свойство Date – введенная дата

5.6. РАБОТА С БАЗАМИ ДАННЫХ В C++Builder

В общем случае схема работы с базой данных следующая:

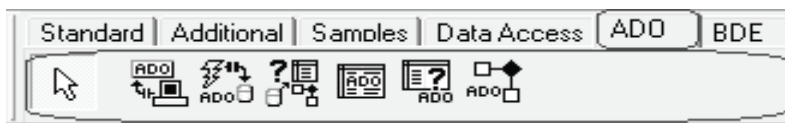


Основной компонент при работе с базами данных – TDataSet – набор данных и его потомки – TTable – таблица БД, TQuery – запрос, TStoredProc – серверная процедура.

Назначение этих компонентов – получение наборов строк из базы данных, а для TQuery и TStoredProc – также выполнение модификации данных. Чтобы работать с базой данных, необходимо прежде всего с ней соединиться. Компонент набора данных может непосредственно соединиться с базой данных, но если в приложении много компонентов наборов данных, целесообразно делать это через специальный компонент соединения (обычно в их названии есть либо Connection, либо Session). Кроме того, компонент соединения позволяет управлять транзакциями на уровне приложения и получать информацию о произошедших ошибках через свойство Errors. К компоненту соединения остальные компоненты привязываются с помощью свойства Connection. После того как компонент TDataSet соединился с базой данных и получил набор строк, нужно сделать эти данные видимыми для пользователя. Этой цели служат компоненты визуализации и управления данными (Data Controls), такие как TDBGrid – табличное отображение курсора, TDBLookupListBox и TDBLookupComboBox – отображение списка значений определенного поля и т. п. Компонент TDataSource действует как посредник между компонентами TDataSet (TTable, TQuery, TStoredProc) и компонентами Data Controls.

5.7. ОБЗОР ADO-КОМПОНЕНТОВ ДЛЯ РАБОТЫ С БД

Для работы с ADO на вкладке компонентов ADO есть шесть компонентов: ADOConnection, ADOCommand, ADODataSet, ADOTable, ADOQuery, ADOStoredProc

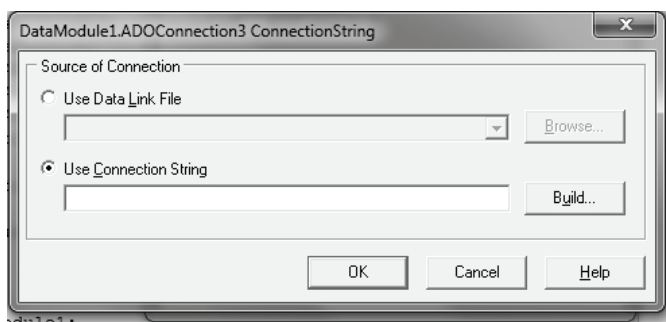


ADOConnection – компонент соединения с базой данных. Используется для соединения с базой данных и работы с транзакциями.

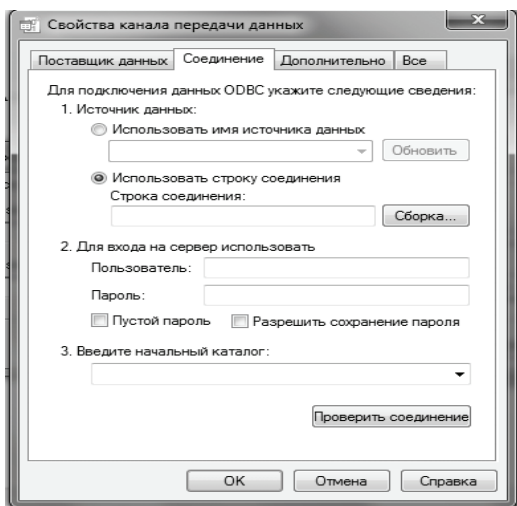
Основное свойство **ADOConnection** – **ConnectionString** – строка соединения с базой данных. Его значение представляет собой набор параметров, разделенных точкой с запятой, их порядок значения не имеет. Выглядит она примерно так:

```
Provider=MSDASQL.1; Persist Security Info=False; User ID=user1;  
Data Source=PostgreSQLst; Mode=Read;
```

Это свойство можно задать вручную, но удобнее использовать стандартный диалог, который вызывается либо двойным кликом на расположенном на форме компоненте **ADOConnection**, либо нажатием на многоточие рядом с пустым полем свойства **ConnectionString** в инспекторе объектов.

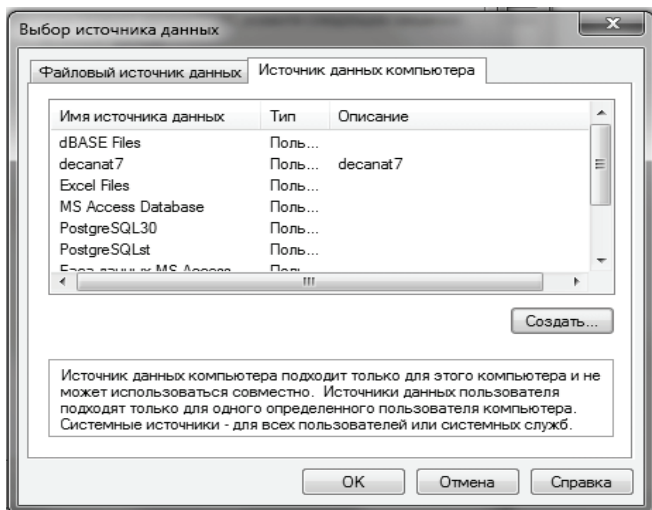


Выбираем «Use Connection String», жмем «Build...».

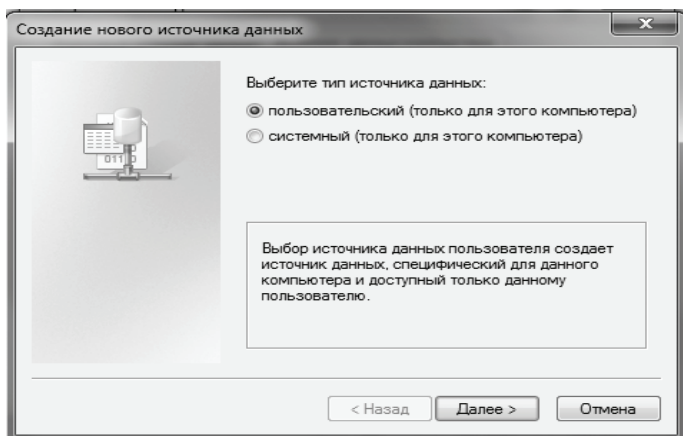


Выбираем закладку «Соединение». Если нужный ODBC-источник был создан ранее, выбираем его в выпадающем списке и жмем «ОК».

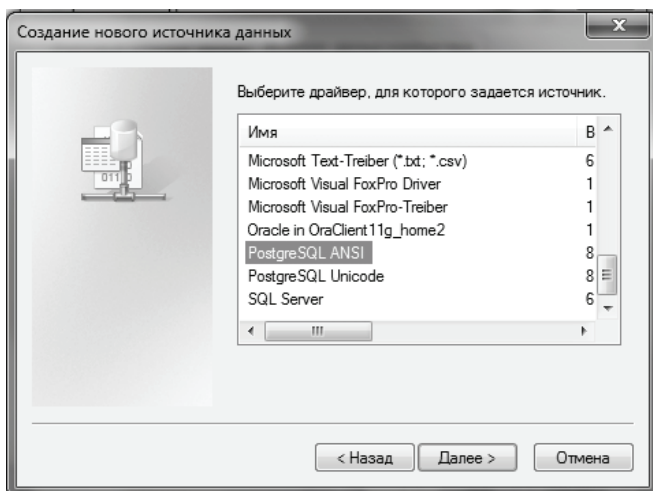
Если нет нужного источника, выбираем «Использовать строку соединения», жмем «Сборка...».

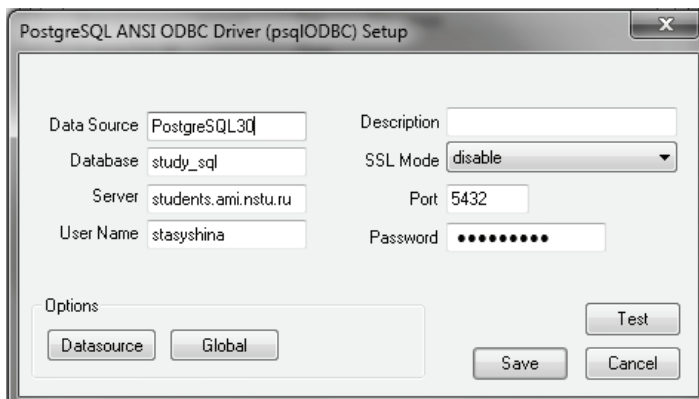


На закладке «Источник данных компьютера» выбираем нужный источник и жмем «ОК», если нужного источника нет, жмем «Создать...».



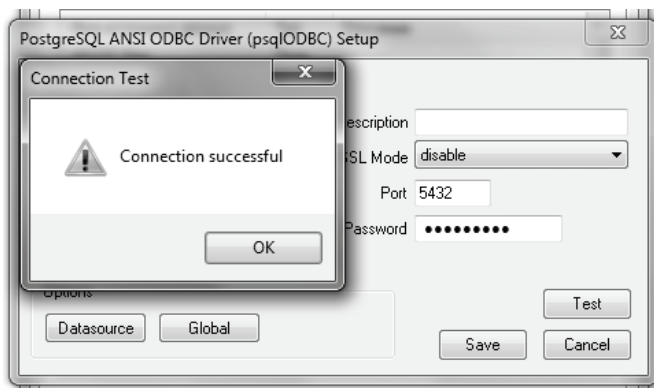
Выбираем драйвер PostgreSQL ANSI (если такого драйвера не оказалось в списке, нужно скачать и установить ODBC драйвер для PostgreSQL версия 8.4 <http://postgresql.ru.net/download.html>).



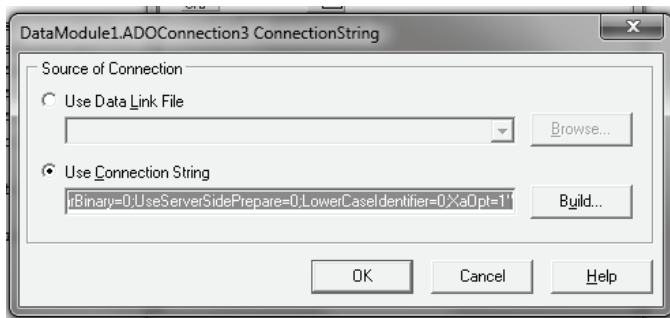


Вводим параметры:

- в поле Data Source – название нашего источника, под которым потом его можно будет выбрать в списке источников;
 - в поле Database – название базы данных (здесь – study_sql);
 - в поле Server – адрес сервера (students.ami.nstu.ru);
 - в поле User Name – логин;
 - в поле Password – пароль.
- Нажимаем кнопку «Test».

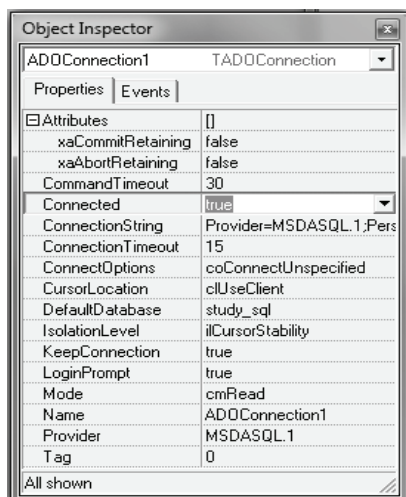


Если соединение прошло успешно, последовательно жмем «OK», «Save», «OK», «OK» и получаем строку подключения. Если соединения не произошло, проверяем правильность ввода параметров.



Теперь можно попытаться соединиться с базой данных.

На этапе проектирования приложения для соединения с базой данных достаточно в инспекторе объектов установить свойство **Connected** в true.



На этапе выполнения приложения для соединения с базой данных используют метод **Open**. Вызов этого метода в программе выглядит так:

```
ADOConnection1->Open();
```

Как правило, его включают в обработчик события OnShow главной формы.

Для разрыва соединения используется метод **Close**.

```
ADOConnection1->Close();
```

Работа с транзакциями

Свойство **IsolationLevel** компонента ADOConnection устанавливает уровень изоляции транзакции. Может принимать одно из следующих значений.

IIUnspecified	Сервер будет использовать лучший, по его мнению, тип изоляции
IIChaos	Транзакции с более высоким уровнем изоляции не могут изменять данные, измененные, но не подтвержденные в текущей транзакции
IIBrowse IIReadUncommitted	Чтение данных, измененных в неподтвержденных транзакциях, т. е. изменения видны сразу, после того как другая транзакция передала их на сервер
IIReadCommitted IICursorStability	Чтение данных, измененных подтвержденными транзакциями, т. е. изменение данных будет видимо после выполнения Commit в другой транзакции
IIRepeatableRead	Изменения, сделанные другими транзакциями, невидимы, но при выполнении перезапроса транзакция может получать новый набор данных
IIIsolated IISerializable	Транзакция не видит изменений данных, произведенных другими транзакциями

Методы для работы с транзакциями компонента ADOConnection:

- **BeginTrans** – начинает транзакцию;
- **CommitTrans** – подтверждает сделанные изменения;
- **RollbackTrans** – откатывает транзакцию.

ADOQuery используется для выполнения запросов к базе данных. Это может быть как запрос, в результате которого возвращаются данные (**SELECT**, возвращенные данные в этом случае всегда рассматри-

ваются как курсор), так и запрос, не возвращающий данных (**INSERT**, **UPDATE**, **DELETE**).

Свойства **ADOQuery**:

- **Connection** – имя компонента ADOConnection, через который осуществляется соединение с базой данных;
- **SQL** – текст запроса;
- **Active** – указывает, открыт (true) или нет (false) курсор (для запросов, возвращающих данные);
- **Eof, Bof** – эти свойства принимают значение true, когда указатель текущей записи расположен на последней или соответственно первой записи курсора;
- **Fields** – массив объектов TField. Используя это свойство, можно обращаться к полям запроса по номеру:

```
Edit1->Text=ADOQuery1->Fields[2]->AsString;
```

Методы компонента **ADOQuery**:

- **Open и Close** устанавливают значения свойства Active равными True и False соответственно. Этот метод используется для запросов, возвращающих курсор;
- **ExecSQL** – выполнение запроса. Этот метод используется для запросов модификации данных;
- **Refresh** позволяет заново считать набор данных из БД;
- **First, Last, Next, Prior** перемещают указатель текущей записи на первую, последнюю, следующую и предыдущую записи соответственно, например:

```
ADOQuery1->First();  
while (!ADOQuery1->Eof)  
{  
    //что-то делаем...  
    ADOQuery1->Next();  
};
```

- **FieldByName** – предоставляет возможность обращения к данным в полях по имени **поля**:

```
S=ADOQuery1->FieldByName(«n_izd»)->AsString;
```

Из событий остановимся на событии **AfterScroll**. Оно происходит каждый раз при переходе от одной записи курсора к другой. Это событие используется, когда нужно, чтобы для каждой записи запроса выполнялось какое-то действие. Например, нужно, чтобы для текущей записи запроса ADOQuery1 выполнялся запрос ADOQuery2. Для этого следует написать обработчик события AfterScroll для ADOQuery1 такого вида:

```
void __fastcall TDataModul::ADOQuery1AfterScroll(TDataSet
*DataSet)
{ ADOQuery2->Close(); // закрываем запрос
//присваиваем параметрам ADOQuery2 значения из текущей стро-
ки ADOQuery1
ADOQuery2->Parameters->ParamValues[«NJ1»]=ADOQuery1-
>FieldByName(«N_IЗD»)->AsString;
ADOQuery2->Parameters->ParamValues[«YR1»]=ADOQuery1-
>FieldByName(«YR»)->AsInteger;
ADOQuery2->Open(); // заново открываем запрос }
```

- **ADOTable** – используется для доступа к одной таблице.

Большинство свойств, методов и событий у ADOTable те же, что у ADOQuery. Основное отличие – вместо свойства SQL у ADOTable есть свойство **TableName** – имя таблицы базы данных. Также у этого компонента есть свойство **ReadOnly** (если это свойство равно true, таблица открыта в режиме «только для чтения») и методы **Insert**, **Edit**, **Delete**, **Append** и **Post**, в совокупности позволяющие изменять данные в таблице.

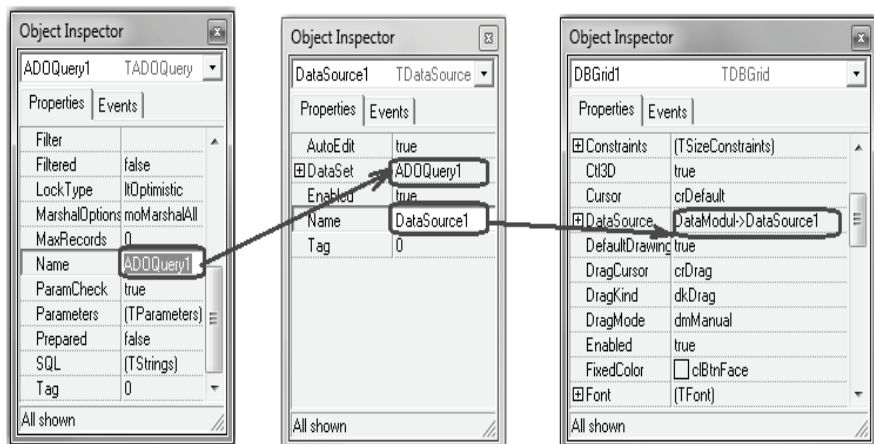
ADOStoredProc – вызов хранимой процедуры. Большинство свойств и методов компонента ADOQuery имеются и у **ADOStoredProc**. Для задания имени процедуры используется свойство **ProcedureName**.

5.8. КОМПОНЕНТЫ ВИЗУАЛИЗАЦИИ ДАННЫХ

DBGrid

Компонент **DBGrid** обеспечивает табличный способ отображения на экране строк данных из компонентов ADODataSet (ADOQuery, ADOTable). Приложение может использовать DBGrid для отображения и редактирования данных БД.

Связь компонента DBGrid с компонентом ADODataset происходит через компонент DataSource. Она осуществляется следующим образом. Свойство DataSet через компонент DataSource устанавливается равным имени компонента ADODataset (как правило, выбирается из выпадающего списка). В свою очередь свойство DataSource компонента DBGrid устанавливается равным имени компонента DataSource.

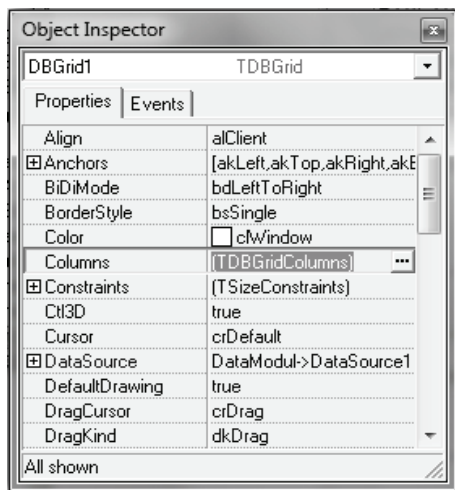


Далее достаточно открыть запрос и данные отобразятся в виде таблицы.

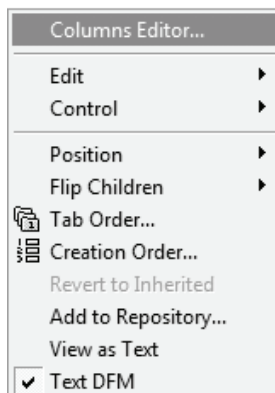
n_izd	yr	kol_det_p	kol_det_isp	kol_det_ost
J1	2011	1600	1050	550
J1	2012		525	25
J1	2014		7	18
J2	2011	2600	780	1820
J2	2012	200	780	1240
J2	2014			1240
J3	2011	700	595	105
J3	2012	1050	770	385
J3	2014			385
J4	2011	6300	2800	3500
J4	2012	500	840	3160
J4	2014			3160
J5	2011	2450	1155	1295
J5	2012		1190	105
J5	2014			105

Внешний вид таблицы может быть изменен.

С помощью редактора свойств Columns Editor можно изменить свойства отдельных столбцов (заголовки, шрифт, цвет фона и т. д.). Для вызова Columns Editor нужно либо выбрать соответствующую опцию в контекстном меню компонента DBGrid, либо щелкнуть мышью в колонке значений напротив свойства Columns в инспекторе объектов.



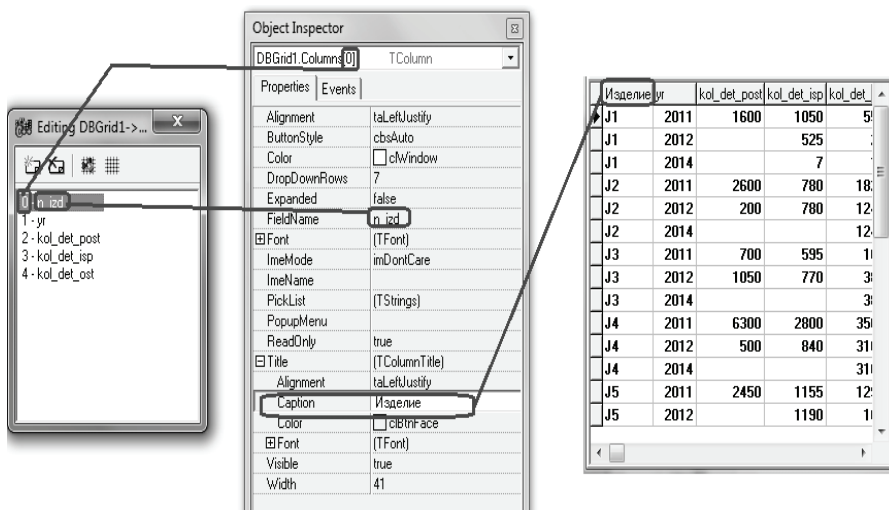
ИЛИ



Появится окно редактора полей. Далее достаточно нажать кнопку Add All Fields и все поля запроса появятся в окне редактора полей

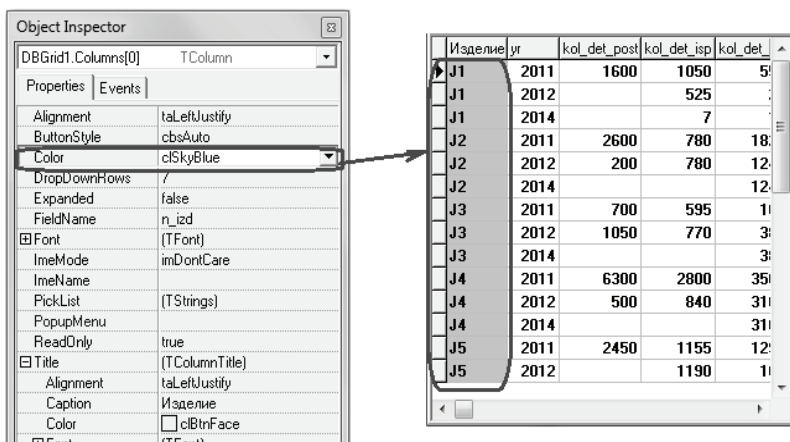


Выделим одно из полей и нажмем клавишу F11, получим доступ к свойствам выбранного поля в инспекторе объектов



В заголовке инспектора объектов отображается номер колонки в таблице, в свойстве FieldName – название поля, которое показывается в этой колонке. Свойство Title отвечает за заголовок, в частности его свойство Caption – текст заголовка колонки.

Меняя свойство Color, можно раскрасить колонки в разный цвет.



Но более сложная и интересная задача – выделить цветом те строки таблицы, которые удовлетворяют некоторому критерию. Для этого можно использовать событие OnDrawColumnCell. Событие OnDrawColumnCell возникает при прорисовке каждой ячейки, при этом текущей записью базового НД становится запись с прорисовываемой ячейкой. Это дает возможность проверить значения полей записи и изменить параметры прорисовки соответствующей ячейки. Напишем для компонента DBGrid обработчик события OnDrawColumnCell. Выделим на форме компонент DBGrid, нажмем клавишу F11 и в инспекторе объектов на странице События (Events) найдем событие OnDrawColumnCell.

Сделав двойной клик на пустом поле рядом с этим событием, перейдем в редакторе кода к тому месту в модуле формы, где нужно написать операторы обработчика. Эти операторы будут примерно такими:

```
// DataModul->ADOQuery1 – компонент, данные которого отображает DBGrid1
// если значение поля kol_det_ost в строке меньше 100
if (DataModul->ADOQuery1->FieldByName(«kol_det_ost»)->AsInteger<100 )
{ // цвет фона строки установить 0x00FEEEC5
  DBGrid1->Canvas->Brush->Color = 0x00FEEEC5;
  // цвет шрифта строки установить clBlue
  DBGrid1->Canvas->Font->Color = clBlue;
  // стиль шрифта строки установить fsBold(жирный)
```

```
    DBGrid1->Canvas->Font->Style=DBGrid1->Canvas->Font->Style<<fsBold;
}
    DBGrid1->Canvas->TextRect(Rect,Rect.Left,Rect.Top,Column->Field->DisplayText);
```

При проектировании на форме ничего не изменится, но при выполнении приложения получится следующая картина:

	Изделие	yr	kol_det_post	kol_det_isp	kol_det_ost
▶	J1	2011	1600	1050	550
	J1	2012		525	25
	J1	2014		7	18
	J2	2011	2600	780	1820
	J2	2012	200	780	1240
	J2	2014			1240
	J3	2011	700	595	105
	J3	2012	1050	770	385
	J3	2014			385
	J4	2011	6300	2800	3500
	J4	2012	500	840	3160
	J4	2014			3160
	J5	2011	2450	1155	1295
	J5	2012		1190	105
	J5	2014			105
	J6	2011	400	390	10
	J6	2012	700	690	20
	J6	2014			20

DBLookupComboBox

Компоненты DBLookup – выпадающий список, связанный с запросом, его удобно использовать, если необходимо организовать выбор некоторого значения из списка, получаемого запросом. Особенно если при этом желательно вывести на экран описательную информацию поля (например, имя поставщика), а после выбора иметь возможность получить соответствующий код (код поставщика).

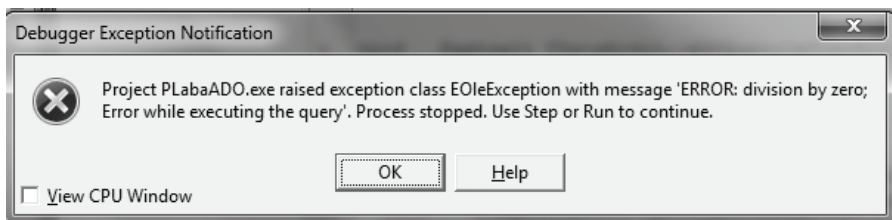
Так же как и компонент DBGrid, DBLookupComboBox связывается с компонентом ADODataset через компонент DataSource. В качестве значения свойства ListSource компонента DBLookupComboBox нужно выбрать имя компонента DataSource, связанного с нужным запросом. Далее в свойстве ListField нужно указать имя поля, которое должно использоваться для выбора, а в свойстве KeyField – имя поля, значение которого будет браться в качестве результата выбора. Компонент готов к работе, главное – не забыть открыть нужный запрос, перед тем как дать пользователю доступ к выбору. Лучше всего это делать до того, как форма, на которой лежит

DBLookupComboBox, станет видимой. Этот момент соответствует событию **OnShow** формы, в обработчик этого события нужно включить оператор вида

```
DataModul->ADOQuery3->Open();
```

5.9. ОБРАБОТКА ОШИБОЧНЫХ СИТУАЦИЙ

При работе программы могут возникать различного рода ошибки: переполнение, деление на нуль, попытка открыть несуществующий файл и т. п. При возникновении таких исключительных ситуаций программа генерирует так называемое исключение (объект класса **Exception**) и дальнейшее выполнение программы прекращается. Исключение – это объект специального вида, характеризующий возникшую в программе особую ситуацию. Он может содержать в виде параметров некоторую уточняющую информацию. Особенностью исключений является то, что это временные объекты. Как только они обработаны каким-то обработчиком, они разрушаются. Если исключение не перехвачено нигде в программе, то оно автоматически обрабатывается методом **TApplication.HandleException**. Он обеспечивает стандартную реакцию программы на большинство исключений – выдачу пользователю краткой информации в окне сообщений и уничтожение экземпляра исключения.



На рисунке приведен пример такого стандартного сообщения для случая целочисленного деления на нуль при выполнении запроса.

Наиболее кардинальный путь борьбы с исключениями – отлавливание и обработка их при помощи блоков **try...catch**. Синтаксис этих блоков следующий:


```

try
{
    Исполняемый код
}
catch (TypeToCatch)
{
    Код, исполняемый в случае ошибки
}

```

Параметр TypeToCatch может быть ссылкой на класс исключений (например, EOverflow – переполнение, EZeroDivide – деление на 0) или многоточием, что означает обработку любого исключения. Блоков catch может быть несколько, для разных типов исключений. Код, исполняемый в блоке, используется для формирования сообщения пользователю и, что более важно, для отката транзакций и освобождения динамически выделенных ресурсов.

Пример обработки ошибочной ситуации:

```

try
{
    // старт транзакции
    DataModul->ADOConnection1->BeginTrans();

    // выполнение процедур и запросов обновления данных
    DataModul->ADOStoredProc1->ExecProc();
    DataModul->ADOQuery2->ExecSQL();

    // подтверждение транзакции
    DataModul->ADOConnection1->CommitTrans();
}
catch (Exception &exception)
{
    // откат транзакции
    DataModul->ADOConnection1->RollbackTrans();
    // сообщение пользователю
    MessageDlg(«При обновлении произошла следующая ошибка:
«+exception.Message, mtError, TMsgDlgButtons() << mbOK, 0);
}

```

6. ПРОЦЕДУРНАЯ ПОДДЕРЖКА ОГРАНИЧЕНИЙ ЦЕЛОСТНОСТИ

Ограничение целостности – это некоторое утверждение, которое может быть истинным или ложным в зависимости от состояния базы данных.

По способам реализации различают:

- *декларативную поддержку ограничений целостности*, выполняемую средствами языка определения данных SQL;
- *процедурную поддержку ограничений целостности*, выполняемую посредством триггеров и хранимых процедур (функций).

Триггер определяет операцию, которая должна выполняться при наступлении некоторого события в базе данных. Триггеры срабатывают при выполнении с таблицей команды SQL INSERT, UPDATE или DELETE.

6.1. СИНТАКСИС ТРИГГЕРОВ И ТРИГГЕРНЫХ ФУНКЦИЙ В PostgreSQL

В PostgreSQL триггеры создаются на основе существующих функций: сначала командой CREATE FUNCTION определяется триггерная функция, затем на ее основе командой CREATE TRIGGER создается триггер. Синтаксис определения триггера:

```
CREATE TRIGGER триггер  
BEFORE | AFTER } { событие [ OR событие ] } ON таблица  
FOR EACH { ROW | STATEMENT }  
EXECUTE PROCEDURE функция (аргументы)
```

Ниже приводятся краткие описания компонентов этого определения.

- **CREATE TRIGGER триггер.** В аргументе *триггер* указывается произвольное имя создаваемого триггера. Имя может совпадать с именем триггера, уже существующего в базе данных, при условии, что

этот триггер установлен для другой таблицы. Кроме того, по аналогии с большинством других несистемных объектов баз данных имя триггера (в сочетании с таблицей, для которой он устанавливается) должно быть уникальным лишь в контексте базы данных, в которой он создается.

- **{ BEFORE | AFTER }.** Ключевое слово BEFORE означает, что функция должна выполняться *перед* попыткой выполнения операции, включая все встроенные проверки ограничений данных, реализуемые при выполнении команд INSERT и DELETE. Ключевое слово AFTER означает, что функция вызывается после завершения операции, приводящей в действие триггер.

- **{ событие [OR событие ...] }.** События SQL, поддерживаемые в PostgreSQL: INSERT, UPDATE или DELETE. При перечислении нескольких событий в качестве разделителя используется ключевое слово OR.

- **ON таблица.** Имя таблицы, модификация которой заданным событием приводит к срабатыванию триггера.

- **FOR EACH { ROW | STATEMENT }.** Ключевое слово, следующее за конструкцией FOR EACH и определяющее количество вызовов функции при наступлении указанного события. Ключевое слово ROW означает, что функция вызывается *для каждой модифицируемой записи*. Если функция должна вызываться всего один раз для всей команды, используется ключевое слово STATEMENT.

- **EXECUTE PROCEDURE функция (аргументы).** Имя вызываемой функции с аргументами. На практике аргументы при вызове триггерных функций не используются.

Синтаксис определения триггерной функции:

```
CREATE FUNCTION функция () RETURNS trigger AS '  
DECLARE  
    объявления ;  
BEGIN  
    команды;  
END;'  
LANGUAGE plpgsql;
```

В триггерных функциях используются специальные переменные, содержащие информацию о сработавшем триггере. С помощью этих переменных триггерная функция работает с данными таблиц.

Ниже перечислены некоторые специальные переменные, доступные в триггерных функциях.

Имя	Тип	Описание
NEW	RECORD	Новые значения полей записи базы данных, созданной командой INSERT или обновленной командой UPDATE, при срабатывании триггера уровня записи (ROW). Переменная используется для модификации новых записей Внимание!!! Переменная NEW доступна только при операциях INSERT и UPDATE. Поля записи NEW могут быть изменены триггером
OLD	RECORD	Старые значения полей записи базы данных, содержащиеся в записи перед выполнением команды DELETE или UPDATE при срабатывании триггера уровня записи (ROW) Внимание!!! Переменная OLD доступна только при операциях DELETE и UPDATE. Поля записи OLD можно использовать только для чтения, изменять нельзя
TG_NAME	name	Имя сработавшего триггера
TG_WHEN	text	Строка BEFORE или AFTER в зависимости от момента срабатывания триггера, указанного в определении (до или после операции)
TG_LEVEL	text	Строка ROW или STATEMENT в зависимости от уровня триггера, указанного в определении
TG_OP	text	Строка INSERT, UPDATE или DELETE в зависимости от операции, вызвавшей срабатывание триггера
TG_RELID	oid	Идентификатор объекта таблицы, в которой сработал триггер.
TG_RELNAME	name	Имя таблицы, в которой сработал триггер

К отдельным полям записей NEW и OLD в триггерных процедурах обращаются следующим образом: NEW.names , OLD.rg.

6.2. ПРАВИЛА ИСПОЛЬЗОВАНИЯ ТРИГГЕРОВ ПРИ ОРГАНИЗАЦИИ ПРОЦЕДУРНОЙ ПОДДЕРЖКИ ОГРАНИЧЕНИЙ ЦЕЛОСТНОСТИ

Ниже на примерах показываются основные требования к разрабатываемым триггерам.

Правило 1. *Триггеры должны реализовать только те ограничения целостности, которые не могут быть заданы декларативными средствами, т. е. с помощью ограничений NOT NULL, UNIQUE, PRIMARY KEY, DEFAULT, CHECK и FOREIGN KEY REFERENCES.*

Будем полагать, что в базе данных библиотеки имеется таблица «Выдача литературы»

```
ATE TABLE delivery1
```

```
(id integer PRIMARY KEY,                -- код выдачи
 id_exemplar integer REFERENCES exemplar(id) -- код экземпляра
                                     книги
 id_reader integer REFERENCES reader(id),   -- код читателя
 id_librarian integer REFERENCES librarian(id), -- код библиотекаря
 delivery_date date NOT NULL                -- дата выдачи
      check ( delivery_date<=now()),
 return_date date check ( return_date <=now())); -- дата возврата
```

Неправильное решение

Триггер, проверяющий ограничение: дата возврата книги не может предшествовать дате выдачи.

```

CREATE FUNCTION tf_delivery_bef_ins_upd () RETURNS SETOF
trigger
LANGUAGE plpgsql
AS $$
BEGIN
    if NEW.return_date < NEW.delivery_date --
    then raise exception 'Дата возврата предшествует дате
выдачи';
        return NULL;
    end if;
    return NEW;
END;
$$;
CREATE TRIGGER trg_delivery_bef_ins_upd
    BEFORE INSERT OR UPDATE ON delivery
    FOR EACH ROW
    EXECUTE PROCEDURE tf_delivery_bef_ins_upd();
Написание данного триггера нецелесообразно, поскольку это ограни-
чение может быть реализовано более простыми декларативными
средствами:
ALTER TABLE ONLY delivery
    ADD CONSTRAINT ddd CHECK ((delivery_date <= return_date));

```

Правильное решение

Триггер, проверяющий ограничение: экземпляр книги не может одновременно находиться более чем у одного читателя. В терминах базы данных это ограничение можно записать следующим образом: среди строк таблицы **delivery1** с одним и тем же значением **id_exemplar**:

- может быть максимум одна строка с пустой датой возврата (**return_date is null**);
- не может быть строк с пересечением периодов с **delivery_date** по **return_date**.

```

CREATE FUNCTION tf_delivery_bef_ins_upd () RETURNS SETOF
trigger
    LANGUAGE plpgsql
    AS $$
BEGIN
if (exists (select id
            from delivery t
            where t.id<>new.id
              and
              t.id_exemplar = new.id_exemplar
              and
              t.delivery_date < case when new.return_date is null
                                   then now()
                                   else new.return_date
                                end
              and
              case when t.return_date is null
                   then now()
                   else t.return_date
              end >new.delivery_date
            )
)
THEN
    raise exception 'Экземпляр одновременно у двух читателей';
    return NULL;
end if;
return NEW;
END;
CREATE TRIGGER trg_delivery_bef_ins_upd
    BEFORE INSERT OR UPDATE ON delivery FOR EACH ROW
    EXECUTE PROCEDURE tf_delivery_bef_ins_upd();

```

Правило 2. Триггеры должны быть написаны для всех операций и всех таблиц, которые затрагивает ограничение.

Ограничение целостности, которое реализует триггер, должно быть четко сформулировано как в терминах бизнес-области, так и в терминах базы данных. При формулировании ограничения в терминах базы данных нужно описать допустимое состояние данных в таблицах вне зависимости от производимых над ними действий, а затем выявить все операции с базой данных, которые могут привести к нарушению ограничения. Триггеры должны быть определены для каждой из этих операций, и не должно быть такой ситуации, что ограничение касается двух таблиц, при этом для одной триггер написан, а для другой – нет.

Предположим, что в базе данных гостиничного комплекса имеются таблицы:

CREATE TABLE hotel	-- Гостиницы
(id_hotel integer PRIMARY KEY,	-- код гостиницы
name_hotel character(20) NOT NULL,	-- название гостиницы
max_floor integer NOT NULL check(max_floor	-- количество этажей
>0),);	
CREATE TABLE room	-- Номера
(id_room integer PRIMARY KEY	-- код номера
id_hotel integer REFERENCES Hotel	-- код гостиницы
(id_hotel),	
number_room character(20) NOT NULL,	-- номер комнаты
max_guest integer NOT NULL	
check(max_guest>0),	-- количество мест
floor integer NOT NULL check(floor >0),	
.....);	-- этаж

Неправильное решение

Проверяются ограничения:

- при добавлении нового номера его этаж должен быть не больше количества этажей в гостинице;
- при вставке новой записи надо проверить, что этаж номера не больше этажности гостиницы.


```

CREATE FUNCTION tf_room_bef_ins() RETURNS SETOF trigger
  LANGUAGE plpgsql
  AS $$
BEGIN
if (new.floor > (select max_floor from hotel t
                  where t.id_hotel=new.id_hotel)
    )
THEN
  raise exception 'У номера слишком высокий этаж';
  return NULL;
end if;
return NEW;
END;
$$;
CREATE TRIGGER trg_room_bef_ins
BEFORE INSERT ON room
FOR EACH ROW
EXECUTE PROCEDURE tf_room_bef_ins()

```

Правильное решение

Ограничение в терминах бизнес-области имеет вид: ни один гостиничный номер не может располагаться на этаже, превышающем этажность гостиницы.

Это ограничение в терминах базы данных трансформируется в следующее: значение поля **floor_room** в строке таблицы **room** не должно быть больше, чем значение поля **max_floor** в строке таблицы **hotel** со значением поля **Id_hotel = room.Id_hotel**.

Нарушение ограничения возможно при следующих операциях: вставке и модификации строк таблицы **room**, модификации строк таблицы **hotel**. Для решения задачи требуются два триггера:

- перед вставкой или обновлением строки в таблице **room**;
- перед обновлением строки в таблице **hotel**.

```

CREATE FUNCTION tf_room_bef_ins() RETURNS SETOF trigger
    LANGUAGE plpgsql
    AS $$
DECLARE
    fl integer;
BEGIN
    select max_floor into fl
    from hotel
    where hotel.id_hotel=new.id_hotel;
    if (new.floor > fl)
    THEN
        raise exception 'У номера слишком высокий этаж. Максимум -
%', fl;
        return NULL;
    end if;
    return NEW;
END;
$$;
CREATE TRIGGER trg_room_bef_ins
BEFORE INSERT OR UPDATE ON room
FOR EACH ROW
EXECUTE PROCEDURE tf_room_bef_ins();
CREATE FUNCTION tf_hotel_bef_upd() RETURNS SETOF trigger
    LANGUAGE plpgsql
    AS $$
BEGIN
    if (exists (select id_room
                from room t
                where t.floor>new.max_floor)
    )
    THEN
        raise exception 'Существует номер с этажом > %',new.max_floor ;
        return NULL;
    end if;
    return NEW;
END;
$$;
CREATE TRIGGER trg_hotel_bef_upd

```

```

BEFORE UPDATE ON hotel
FOR EACH ROW
EXECUTE PROCEDURE tf_hotel_bef_upd();

```

Правило 3. При нарушении ограничения триггер должен формировать достаточно информативное сообщение об ошибке.

Предположим, что в базе данных библиотеки имеется таблица «Фактические полеты».

```

CREATE TABLE  realrun
(id integer PRIMARY KEY,           -- код полета
 id_flight nteger REFERENCES flight (id), -- код рейса
 id_plane integer REFERENCES plane (id), -- код самолета
 id_team integer REFERENCES team (id),  -- код экипажа
 begin_time timestamp with time zone -- фактические дата-время
NOT NULL check ( begin_time<= cur-   вылета
rent_timestamp),
 end_time timestamp with time zone ,  -- фактические дата-время
                                     приземления
.....
check ( begin_time < end_time ));

```

Необходимо разработать триггер, который проверяет ограничение: самолет не может одновременно выполнять более чем один полет.

В терминах базы данных это ограничение имеет вид: среди строк таблицы **realrun** с одним и тем же значением **id_plane**:

- может быть максимум одна строка с пустым временем приземления (**end_time is null**);
- не может быть строк с пересечением периодов с **begin_time** по **end_time**.

Неправильное решение

```
CREATE FUNCTION tf_realrun_bef_ins_upd ()RETURNS SETOF trigger
LANGUAGE plpgsql
AS $$
DECLARE
new_et timestamp;
BEGIN
new_et:= case when new.end_time is null
            then current_timestamp + interval'1 hour'
            else new.end_time
        end;
if (exists (select id
            from realrun t
            where t.id<>new.id
                and
                t.id_plane = new.id_plane
                and
                t.begin_time < new_et
            and
            case when t.end_time is null
                then current_timestamp
                else t.end_time
            end >new.begin_time
        )
) OR
(exists (select id
        from realrun t
        where t.id<>new.id
            and
            t.id_team = new.id_team
            and
            t.begin_time new_et
        and
        case when t.end_time is null
            then current_timestamp
            else t.end_time
        end >new.begin_time
    )
)
THEN
```

```
raise exception 'ERROR: incorrect data';
return NULL;
end if;
return NEW;
```

Правильное решение

```
CREATE FUNCTION tf_realrun_bef_ins_upd () RETURNS SETOF trigger
LANGUAGE plpgsql
AS $$
DECLARE
new_et timestamp;
BEGIN
new_et:= case when new.end_time is null
then current_timestamp + interval'1 hour'
else new.end_time
end;
if (exists (select id
from realrun t
where t.id <> new.id
and
t.id_plane = new.id_plane
and
t.begin_time < new_et
and
case when t.end_time is null
then current_timestamp + interval'1 hour'
else t.end_time
end > new.begin_time
)
)
THEN
raise exception 'Самолет не может выполнять более одного полета
одновременно';
if (exists (select id
from realrun t
where t.id <> new.id
and
t.id_team = new.id_team
```

```

        and
        t.begin_time < new_et
    and
    case when t.end_time is null
        then current_timestamp + interval'1 hour'
        else t.end_time
    end > new.begin_time
)
)
THEN
    raise exception 'Экипаж не может выполнять более одного полета
одновременно';
    return NULL;
end if;
return NEW;
END;
$$

```

Правило 4. Триггеры не должны компенсировать недостатки проектирования базы данных.

Если требуется большое количество триггеров для поддержания согласованности данных, это, как правило, свидетельствует о недостаточно хорошо спроектированной и/или ненормализованной структуре базы данных.

Например, об этом свидетельствуют внешние ключи с опцией **on delete cascade** и одновременно использование запрещающего триггера.

Правило 5. Триггеры не должны быть бессмысленными и бесполезными.

Пример бессмысленного триггера из студенческого проекта.

Предполагается, что в базе данных имеются следующие таблицы:

CREATE TABLE mu	-- Военная часть
(id integer PRIMARY KEY,	-- код военной части
name_mu character(20) NOT NULL,	-- название военной части
.....);	
CREATE TABLE soldier	-- Военнослужащий
(id integer PRIMARY KEY,	-- код военнослужащего
fio character(50) NOT NULL,	-- фιο
db date NOT NULL,);	-- дата рождения

CREATE TABLE post	-- должность
(id integer PRIMARY KEY,	-- код должности
name_post character(20) NOT NULL,	-- название должности
cansel_date date,	-- дата, с которой долж-
.....);	ность не актуальна
CREATE TABLE job	-- назначение на должность
(id integer PRIMARY KEY,	-- код записи
id_soldier integer NOT NULL	-- код военнослужащего
REFERENCES soldier (id),	
id_mu integer NOT NULL REFERENCES	-- код военной части
mu (id),	-- код должности
id_post integer NOT NULL REFERENCES	-- дата назначения
post (id),	-- дата перевода/увольнения
date_post date NOT NULL,	
date_cansel date);	

Описание триггера: триггер при удалении записи из таблицы «Должность» ставит в столбец «cansel_date» текущую дату, тем самым помечая запись как неиспользуемую.

```

CREATE FUNCTION tf_post_aft_del() RETURNS SETOF trigger
LANGUAGE plpgsql
AS $$
BEGIN
OLD.cansel_date:=current_date;
insert into post values (OLD.id, OLD. name_post, OLD.cansel_date);
return OLD;
END;
$$;
CREATE TRIGGER trg_post_aft_del
AFTER DELETE ON post
FOR EACH ROW
EXECUTE PROCEDURE tf_post_aft_del();

```

Рассмотрим, как будет работать этот триггер.

Возможны два варианта развития событий.

1. Из таблицы **post** удаляется запись, на которую есть хотя бы одна ссылка в таблице **job**. В этом случае сработает ограничение внешнего ключа, так как ограничение внешнего ключа проверяется перед удалением записи. Оно не позволит удалить запись, а поскольку запись не будет удалена, триггер не будет выполнен. Этот вариант наиболее вероятен, так как предполагается, что на основную массу строк таблицы **post** в таблице **job** будут ссылки.

2. Если из таблицы **post** удаляется запись, на которую нет ни одной ссылки в таблице **job**, триггер отработает и восстановит только что удаленную запись и поставит в столбец «**cansel_date**» текущую дату. Получается, что должность, на которую до настоящего момента никто ни разу не был назначен и которая с текущего момента не актуальна (фактически совершенно ненужная должность), будучи однажды вставлена в таблицу, не может быть из нее удалена.

Вывод: это не просто бесполезный, а в какой-то мере даже вредный триггер.

В данной ситуации нужен не триггер, а процедура удаления должности, которая будет проверять, можно ли удалить должность. Если можно, будет ее удалять, если нельзя (когда есть ссылки), будет помечать запись как более не используемую.

```
CREATE FUNCTION delete_post1(inid_post integer) RETURNS void
LANGUAGE plpgsql
AS $$
BEGIN
delete from post where id=inid_post;
exception
when integrity_constraint_violation then
    update post set cansel_date=current_date
    where id=inid_post and cansel_date is null;
when others then
    raise exception 'error NUM:%, DETAILS:%', SQLSTATE,
SQLERRM;
END;
$$;
```

Кроме того, рекомендуется в приложении для операций удаления запрашивать дополнительное подтверждение, чтобы минимизировать возможность случайного ошибочного удаления данных.

7. АНАЛИЗ ДАННЫХ С ИСПОЛЬЗОВАНИЕМ СВОДНЫХ ТАБЛИЦ Microsoft Excel

OLAP-технология представляет для анализа данные в виде многомерных (и, следовательно, нереляционных) наборов данных, называемых многомерными кубами (гиперкуб, метакуб, куб фактов), оси которых содержат параметры, а ячейки – зависящие от них агрегатные данные.

При этом гиперкуб является концептуальной логической моделью организации данных, а не физической реализацией их хранения, храниться данные могут как в многомерном представлении (MOLAP – Multidimensional OLAP), так и в виде плоских реляционных таблиц. В последнем случае предпочтительным является хранение данных в специальном виде (модель ROLAP – Relational OLAP), хотя возможно и использование связанных между собой таблиц, составляющих основу OLTP-приложений.

Осями многомерной системы координат служат основные атрибуты анализируемого бизнес-процесса, по которым проводится анализ. Например, для продаж это могут быть тип товара, регион, тип покупателя. В качестве одного из измерений используется время. На пересечениях осей-измерений (называемых *dimensions*) находятся данные, количественно характеризующие процесс – меры (*measures*): суммы и иные агрегатные функции (*min*, *max*, *avg*, дисперсия и пр.).

Весь набор программных средств для анализа данных можно классифицировать следующим образом.

1. Клиентские OLAP-средства. Как правило, OLAP-функциональность таких OLAP-продуктов реализована в средствах статистической обработки данных (из продуктов этого класса на российском рынке широко распространены продукты компаний StatSoft и SPSS) и в некоторых электронных таблицах. В частности, неплохими средствами многомерного анализа обладает Microsoft Excel, начиная с версии 2000. С помощью этого продукта можно создать и сохранить в виде

файла небольшой локальный многомерный OLAP-куб, отображая его двух- или трехмерные сечения.

2. OLAP-серверы. Клиентские приложения могут запрашивать требуемое многомерное представление и в ответ получать те или иные данные. При этом OLAP-серверы могут хранить многомерные данные разными способами, скрывая от конечного пользователя способ реализации многомерной модели.

3. В последнее время все большую популярность приобретают OLAP-системы, относящиеся к категории DOLAP (Desktop OLAP). Это развитые OLAP-системы, осуществляющие выборку данных из исходных источников (реляционные и многомерные базы данных, электронные таблицы и пр.), преобразующие и помещающие их в динамическую многомерную базу данных, функционирующую на клиентской станции конечного пользователя. Построенный куб данных анализируется средствами многомерного OLAP-анализа на машине клиента.

В данном разделе рассматриваются возможности проведения OLAP-анализа данных средствами так называемых сводных таблиц Microsoft Excel. Схема подготовки данных для их последующего анализа средствами Microsoft Excel представлена на рисунке.

Будем при этом полагать, что данные для анализа хранятся в базе данных PostgreSQL.

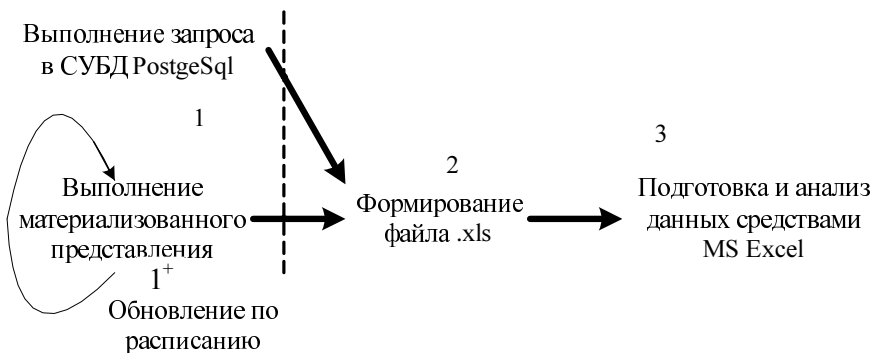


Схема подготовки данных для анализа средствами Microsoft Excel

7.1. ПОДГОТОВКА РЕЗУЛЬТАТОВ БАЗОВОГО ЗАПРОСА ДЛЯ ПОСЛЕДУЮЩЕГО АНАЛИЗА

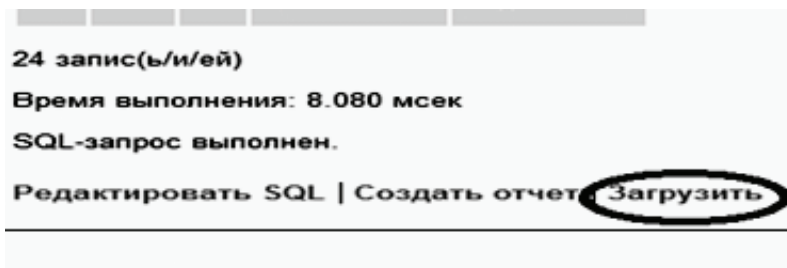
В случае использования данных, хранящихся в модели ROLAP, запрос для последующего анализа имеет вид

```
select  измерение_1,  
        измерение_2,....,  
        измерение_n,  
        sum либо count (мера_1)  мера_1,  
        sum либо count (мера_2)  мера_2,  
        .....  
        sum либо count (мера_m)  мера_m  
from  таблица_фактов  
group by измерение_1,  
        измерение_2,....,  
        измерение_n
```

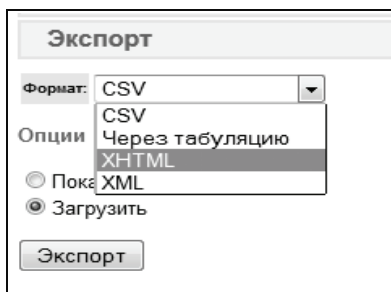
В случае обычной OLTP-базы данных запрос для последующего анализа может иметь произвольный вид.

Первым этапом подготовки данных является сохранение результата запроса в виде Excel-файла. Ниже приведена последовательность шагов по сохранению результата запроса, полученного в рамках PhpPgAdmin, в Excel-файле.

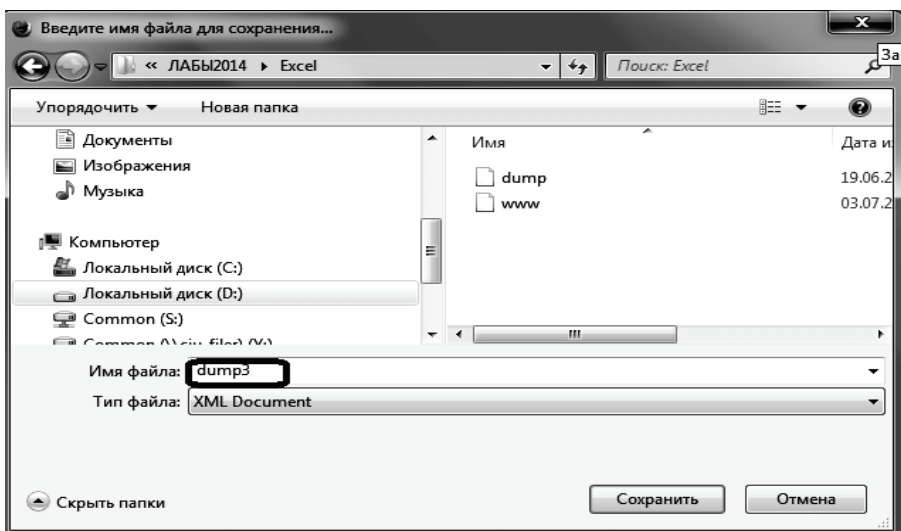
1. Выбрать опцию «Загрузить» на панели PhpPgAdmin



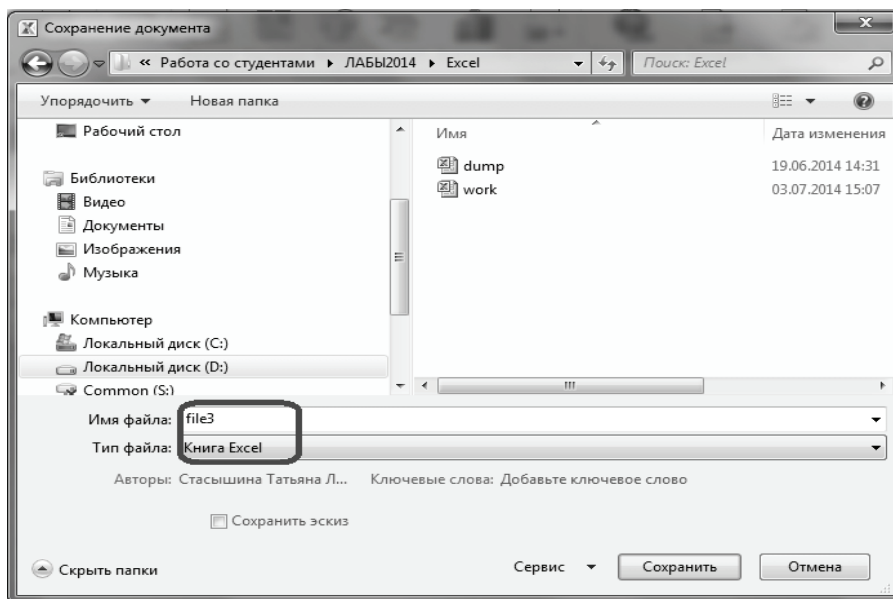
2. Выбрать формат XHTML, опцию «Загрузить» и нажать «Экспорт».



3. В диалоге сохранения указать, где и под каким именем сохранить файл, например, dump3.html.



4. Открыть сохраненный файл dump3.html в Microsoft Excel и пере-сохранить его как файл Excel (например, под именем file3.xls).

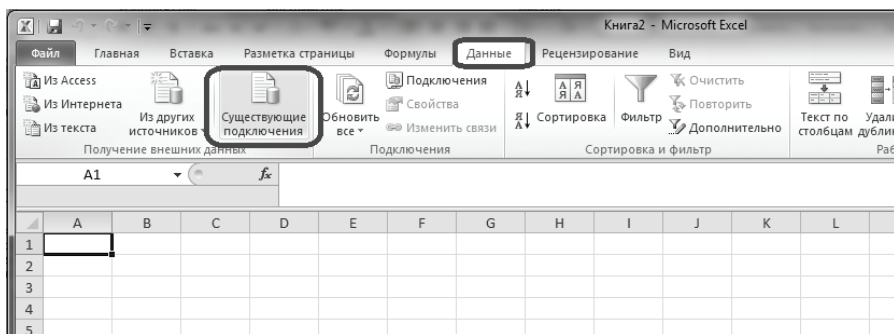


Замечание. Если базовый запрос выполняется достаточно долго, но при этом нет необходимости отслеживать изменение данных в реальном масштабе времени при выполнении аналитических запросов, для базового запроса может быть создано материализованное представление с периодом обновления, соответствующим тому, как часто изменения данных должны отражаться в аналитических запросах. О создании материализованных представлений можно прочитать в литературе по языку SQL PostgreSQL.

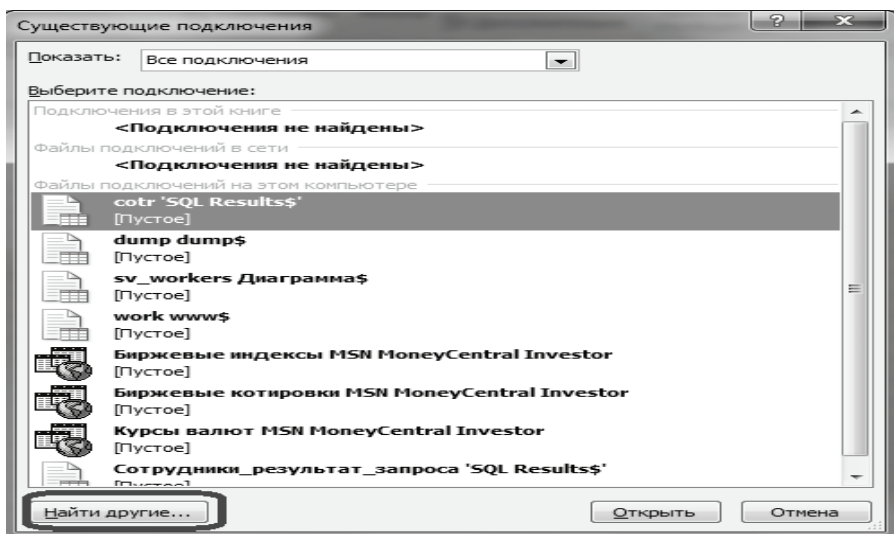
Инструментом Microsoft Excel для анализа данных являются так называемые сводные таблицы.

Последующие шаги связаны с созданием сводной таблицы.

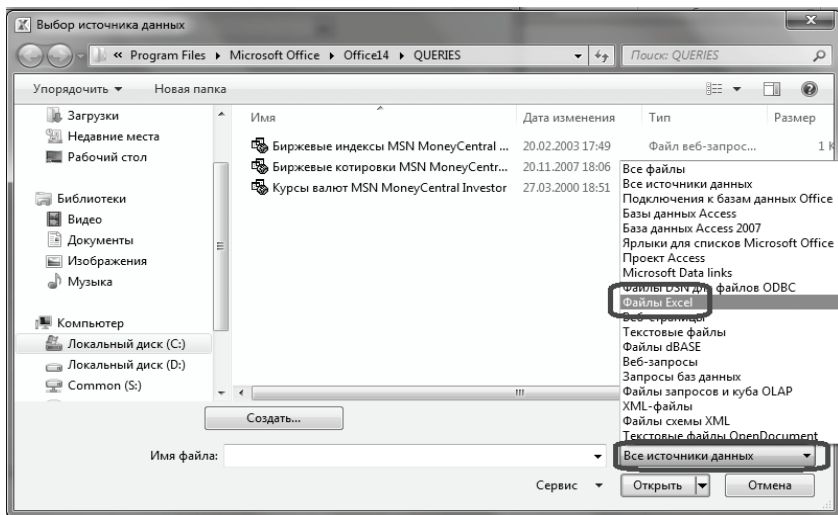
5. Выбрать в меню Microsoft Excel последовательно разделы «Данные»/«Существующие подключения».



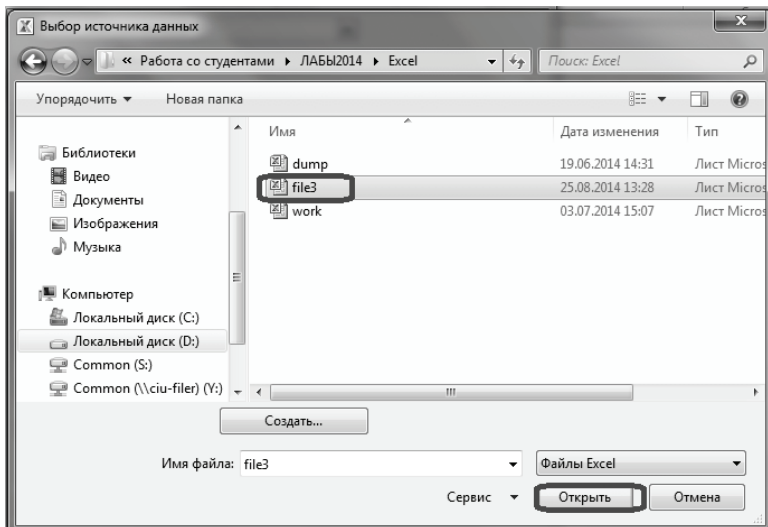
6. В открывшемся диалоге нажать кнопку «Найти другие» (снизу слева).



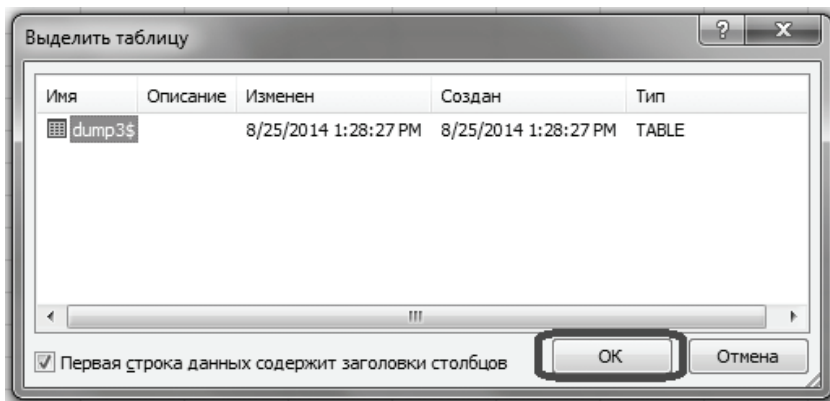
7. В появившемся диалоговом окне «Выбор источника данных» в выпадающем списке «Все источники данных» (в правом нижнем углу) выбрать «файлы Excel».



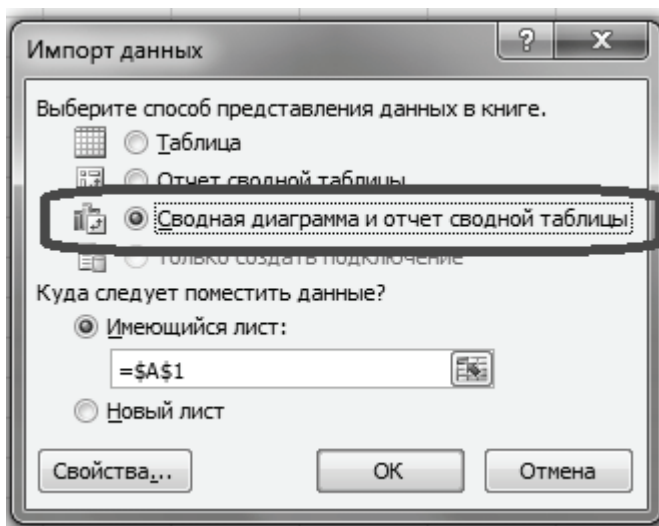
8. Найти сохраненный файл file3.xls, открыть его.



нажав ОК.



9. В диалоговом окне «Импорт данных» выбрать опцию «Отчет сводной таблицы» (или «Сводная диаграмма и отчет сводной таблицы»).



Отчет готов для анализа.

Файл

Главная

Вставка

Разметка страницы

Формулы

Данные

Рецензирование

Вид

Параметры

Конструктор

Имя:

Активное поле:

Активное поле

СводнаяТаблица1

Параметры

Параметры поля

Свернуть все поле

Активное поле

Группа по выделенному

Разгруппировать

Группировка по полю

Группировать

Сортировка

Сортировка и фильтр

Вставить срез

Обновить источник данных

Данные

Очистить

Выделить

Переместить

Действия

Итого по

Дополнительные вычисления

Поля, элементы и наборы

Вычисления

Сводная диаграмма

Книга1 - Microsoft Excel

A1

fx

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P								
1		Названия столбцов																						
2		Сумма по полю Количество	Сумма по полю Марка					Итого Сумма по полю Количество					Итого Сумма по полю Марка											
3	Названия строк	2106	2114	2401	2410	4169	4199	2106	2114	2401	2410	4169	4199											
4	BA3	70	45					4212	6342					115	10554									
5	Белый	50						2106						50	2106									
6	Зеленый	25						2114						25	2114									
7	Красный	20	10					2106	2114					30	4220									
8	Черный	10						2114						10	2114									
9	GA3	210	15					7203	2410					225	9613									
10	Белый	80						2401						80	2401									
11	Красный	30						2401						30	2401									
12	Черный	100	15					2401	2410					115	4811									
13	UA3	55	300					12507	4199					355	16706									
14	Зеленый	10	300					4169	4199					310	8368									
15	Черный	45						8338						45	8338									
16	Общий итог	70	45	210	15	55	300	4212	6342	7203	2410	12507	4199	695	36873									
17																								
18																								

7.2. ПРОВЕДЕНИЕ АНАЛИЗА ДАННЫХ

Чаще всего при проведении OLAP-анализа рассматривают следующие операции:

- операция Сечение (срез – Slice): формируется подмножество гиперкуба, в котором значение одного или более измерений фиксировано (значение параметров для фиксированного, например, месяца);
- операция Вращение (Rotate): меняется порядок представления измерений, обеспечивая представление метакуба в более удобной для восприятия форме;
- операция Консолидация (Drill up): обобщающие операции, как простое суммирование значений (свертка) или расчет с использованием сложных вычислений, включающих другие связанные данные. Например, показатели для отдельных компаний могут быть просто просуммированы с целью получения показателей для каждого города, а показатели для городов могут быть «свернуты» до показателей по отдельным странам;
- операция Спуска (Drill down): операция, обратная консолидации, которая включает отображение подробных сведений для рассматриваемых консолидированных данных;

- операция Разбиение с поворотом (Slicing and dicing): позволяет получить представление данных с разных точек зрения. Например, один срез данных о доходах может содержать **все сведения** о доходах от продаж товаров указанного типа по каждому городу. Другой срез может представлять данные о доходах отдельной компании в каждом из городов.

Возможности для анализа данных продемонстрированы ниже на примере аналитического отчета, используемого в Информационной системе НГТУ.

В корпоративной части портала НГТУ представлено несколько отчетов:



НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Информационная система университета / Аналитические отчеты

Аналитические отчеты

Динамика подачи заявлений

Сводный отчет по поданным заявлениям

Сводный отчет по кадровому составу

Сводный отчет по составу абитуриентов

Сводный отчет по студенческому составу

Сводный отчет об успеваемости по математике

Сводный отчет по успеваемости по физике

Инструкция по работе со сводными таблицами

Отчет по успеваемости составлен на основе приведенного ниже запроса, оформленного как представление view:

```
create or replace view rep_marks_matem as
select f.shortname «ФАКУЛЬТЕТ»,
o.okso_key||'.'||o.okso_key2 «НАПРАВЛЕНИЕ»,
sg.NAME «ГРУППА»,
kc.shortname «КАФЕДРА»,
p.lastname||' '||p.sname||' '||p.fathername «ПРЕПОДАВАТЕЛЬ»,
d.dname «ПРЕДМЕТ»,
```

```

'Кредиты: '||sm.credits||', Л. '||sm.nedel*sm.lect||' Пр.'||sm.prac*sm.nedel
«РАСЧАСОВКА»,
nvl(nvl(sm.nedel,1)*nvl(sm.lect,sm.z_lect),0)+nvl(nvl(sm.prac,sm.z_prac)*
nvl(sm.nedel,1),0) «АУДИТОРНЫЕ ЧАСЫ»,
coalesce(sm.nedel*sm.lect,sm.z_lect) «ЛЕКЦИИ»,
coalesce(sm.prac*sm.nedel,sm.z_prac) «ПРАКТИКИ»,
case when sr.mark < 3 then '2. неуд' when sr.mark = 3 then '3. уд.' when
sr.mark = 4 then '4 хоп.' when sr.mark = 5 then '5. отл' end «ОЦЕНКА»,
case when rs.score < 40 then '1. <40' when rs.score between 40 and 60 then
'2. 40 - 60 ' when rs.score between 61 and 80 then '3. 60 - 80' when rs.score
> 80 then '4. > 80' when rs.score is null then '5. нет' end «БАЛЛ»,
case when rs.fk_type_exam = 24 then 'ЕГЭ' when rs.fk_type_exam = 1 then
'НГТУ' end «ТИП»,
cm.year_enr «ГОД НАБОРА»,
sr.semester «СЕМЕСТР»,
case when sg.FK_TRAINING_FORMS = 1 then 'очная' else 'заочная' end
«ФОРМА»
from student_results sr
join student s on s.pk = sr.fk_student
join register r on r.pk = sr.fk_register
join vw$study_group sg on sg.pk = s.fk_study_group
join armuser.sg_curriculum sgc on sgc.fk_study_group = sg.pk
join sp sp on sp.id_discname = sr.fk_facultet_discipline
join usp u on u.id_usp = sp.id_usp
join discvkaf dk on dk.id_sp = sp.id_sp
join kadry.chair kc on kc.id = dk.id_kafedr
join facultet_okso fo on fo.pk = sg.FK_FACULTET_OKSO
join okso o on o.pk = fo.fk_okso
join curriculum c on c.pk = u.fk_curriculum and c.pk = sgc.fk_curriculum
join kadry.person p on p.id = r.fk_emp
join semestr sm on sm.id_usp = u.id_usp and sm.nsem = sr.semester
join discname d on d.id_discname = sr.fk_facultet_discipline
join facultet f on f.pk = sg.FSK_FACULTET

```


2. На листе «Поля» Excel-файла для справки приведены описания полей.

Поля - измерения									
Наименование	Значение								
И_СТУДЕНТОВ	Количество студентов, соответствующих выбранным характеристикам								
Поля - характеристики									
Наименование	Значение								
Академическая стипендия	Получает ли академическую стипендию (да/нет)								
Гражданство									
Группа									
Кафедра	Выпускающая кафедра								
Квалификация	Код квалификации (62 - бакалавры, 65 - специалисты, 68 магистранты)								
Курс									
Направление	Шифр направления								
Непереведенные	Является ли студент все еще не переведенным на нужный курс (да/нет)								
Основа обучения	Бюджет или контракт								
Повторно	Является ли обучающимся повторно								
Регион	Регион России либо государство, откуда прибыл студент								
Состояние	учится, находится в ак. отпуске								
Социальная стипендия	Получает ли социальную стипендию (да/нет)								
Средний балл ЕГЭ	В каком диапазоне среднего балла ЕГЭ студент поступил								
Средний балл сессий	Диапазон среднего балла по итогам прошедших сессий								
УГС	Укрупненная группа специальностей								
Факультет									
ФГОСЗ	Является ли студент обучающимся по программам ФГОСЗ (да/нет)								

3. Управление полями производится в области «Список полей сводной таблицы».

Список полей сводной таблицы

Выберите поля для добавления в отчет:

☐ АКАДЕМИЧЕСКАЯ СТИПЕНДИЯ
☐ ГРАЖДАНСТВО
☐ ГРУППА
☒ И_СТУДЕНТОВ
☐ КАФЕДРА
☐ КВАЛИФИКАЦИЯ
☒ КУРС
☐ НАПРАВЛЕНИЕ
☐ НЕПЕРЕВЕДЕННЫЕ
☐ ОСНОВА ОБУЧЕНИЯ
☐ ПОВТОРНО

Перетащите поля между указанными ниже областями:

Фильтр отчета

ФОРМА ОБУЧЕНИЯ

Названия столбцов

КУРС

Названия строк

ФАКУЛЬТЕТ

Значения

Сумма по полю И...

☐ Отложить обновление макета

Обновить

4. Для добавления значений характеристики в строки название нужного поля необходимо перенести в раздел «Названия строк».

	A	B	C	D	E	F	G	H	I	J	K	L		
1														
2	ФОРМА ОБУЧЕНИЯ	о	т											
3														
4	Сумма по полю И_СТУДЕНТОВ	Названия столбцов												
5	Названия строк		1	2	3	4	5	6	7	13	15	18	Общий итог	
6	⇒АВТО		363	301	242	205	145	3	2			2	1	1264
7	бюджетная основа обучения		296	248	185	149	86					1	1	966
8	контрактная основа обучения		67	53	57	56	59	3	2					298
9	⇒ИДО		14	25	15	35								89
10	контрактная основа обучения		14	25	15	35								89
11	⇒ИСР		55	60	59	30								225
12	бюджетная основа обучения		53	55	56	28								213
13	контрактная основа обучения		2	5	3	2								12
14	⇒МТО		217	145	128	143								710
15	бюджетная основа обучения		190	125	108	106								569
16	контрактная основа обучения		27	20	20	37								141
17	⇒РЭФ		305	262	193	190								1057
18	бюджетная основа обучения		285	235	156	162								921
19	контрактная основа обучения		20	27	37	28								136
20	⇒ФБ		432	211	185	198						1		1236
21	бюджетная основа обучения		78	72	24	49						1		255
22	контрактная основа обучения		354	139	161	149								981
23	⇒ФОГО		331	219	224	327								1342

Список полей сводной таблицы

Выберите поля для добавления в отчет:

☐ КАФЕДРА
☐ КВАЛИФИКАЦИЯ
☒ КУРС
☐ НАПРАВЛЕНИЕ
☐ НЕПЕРЕВЕДЕННЫЕ
☒ ОСНОВА ОБУЧЕНИЯ
☐ ПОВТОРНО
☐ РЕГИОН
☐ СОСТОЯНИЕ
☐ СОЦИАЛЬНАЯ СТИПЕНДИЯ
☐ СРЕДНИЙ БАЛЛ ЕГЭ

Перетаскивайте поля между указанными ниже областями:

Фильтр отчета

ФОРМА ОБУЧЕНИЯ

КУРС

Названия столбцов

Названия строк

ФАКУЛЬТЕТ

ОСНОВА ОБУЧЕНИЯ

Сумма по полю И...

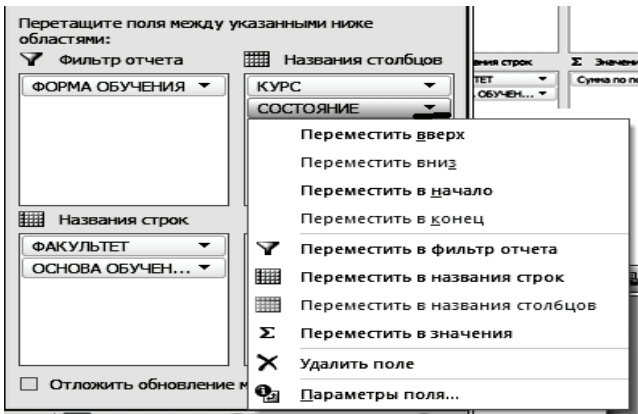
☐ Отложить обновление макета

Обновить

5. Аналогично для добавлений значения характеристики в столбцы название нужного поля необходимо перенести в раздел «Названия столбцов».

ФОРМА ОБУЧЕНИЯ	о	т				
Сумма по полю И_СТУДЕНТОВ	Названия столбцов					
Названия строк	ак. отпуск	учится	1 Итого	2	2 Итого	
АВТО		9 354 363	12	289	301	
бюджетная основа обучения		3 293 296		9 239	248	
контрактная основа обучения		6 61 67		3 50 53		
ИДО		14 14		25 25		
контрактная основа обучения		14 14		25 25		
ИСР		1 54 55		1 59 60		
бюджетная основа обучения		1 52 53		1 54 55		
контрактная основа обучения		2 2		5 5		
МТО		3 214 217		2 143 145		
бюджетная основа обучения		2 188 190		2 123 125		
контрактная основа обучения		1 26 27		20 20		
РЭФ		19 286 305		5 257 262		
бюджетная основа обучения		19 266 285		4 231 235		
контрактная основа обучения		20 20		1 26 27		
ФБ		2 430 432		211 211		
бюджетная основа обучения		78 78		72 72		
контрактная основа обучения		2 352 354		139 139		
ФОГО		5 326 331		7 211 219		
бюджетная основа обучения		2 12 14		1 19 20		
контрактная основа обучения		3 314 317		6 192 199		
ФЛА		7 308 315		2 192 194		
бюджетная основа обучения		6 221 227		2 158 160		
контрактная основа обучения		1 87 88		34 34		

6. Чтобы убрать характеристику, поменять порядок расположения, необходимо щелкнуть по значку раскрытия списка справа от ее названия и выбрать необходимое действие.



7. Чтобы все данные сводной таблицы были отфильтрованы по определенному значению характеристики или набора характеристик, их нужно переместить в раздел «Фильтр отчета» и выбрать необходимые значения из раскрывающегося списка в левой верхней части сводной таблицы.

ФОРМА ОБУЧЕНИЯ

Поиск

(Все)

3

3к

3у

6

оз

озу

☐ Выделить несколько элементов

OK

Отмена

	1 Итого	2	2 Итого	
учится	ак. отпуск	учится		
354	363	12	289	301
293	296	9	239	248
61	67	3	50	53
14	14		25	25
14	14		25	25
54	55	1	59	60
52	53	1	54	55
2	2		5	5
214	217	2	143	145
2	188	190	2	123
1	26	27		20
19	286	305	5	257
19	266	285	4	231
	20	20	1	26
2	430	432		211
	78	78		72
2	357	354		139

Выберите поля для добав

☐ ГРУППА

☒ И. СТУДЕНТОВ

☐ КАФЕДРА

☐ КВАЛИФИКАЦИЯ

☒ КУРС

☐ НАПРАВЛЕНИЕ

☐ НЕПЕРЕВЕДЕННЫЕ

☒ ОСНОВА ОБУЧЕНИЯ

☐ ПОВТОРНО

☐ РЕГИОН

☒ СОСТОЯНИЕ

☐ СОЦИАЛЬНАЯ СТИПЕНДИЯ

☐ СРЕДНИЙ БАЛЛ ЕГЭ

☐ СРЕДНИЙ БАЛЛ СЕССИИ

☐ УГС

☒ ФАКУЛЬТЕТ

☐ ФГОСЗ

Перетащите поля между областями:

Фильтр отчета

ФОРМА ОБУЧЕНИЯ

Осуществляя перечисленные в пп. 4–7 действия, пользователь выполняет перечисленные выше OLAP-операции.

Все изменения в настройках сводной таблицы сохраняются в файле на компьютере, на котором проводится анализ. Для получения обновленных данных необходимо повторно скопировать файл с портала Информационной системы. Обновление данных производится раз в неделю.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Хансен Г. Базы данных и управление / Г. Хансен, Д. Хансен. – М.: Бином, 1999.
2. Дейт К. Введение в системы баз данных / К. Дейт. – М.: Вильямс, 2000.
3. Грофф Д.Р. SQL: Полное руководство / Д.Р. Грофф, П.Н. Вайнберг. – Киев: BMV, «Ирина», 2001.
4. Карпова Т.С. Базы данных: модели, разработка, реализация: учеб. пособие / Т.С. Карпова. – СПб.: Питер, 2001.
5. Конноли Т. Базы данных. Проектирование, реализация и сопровождение / Т. Конноли, К. Бегг, А. Страчан. – М.; СПб.; Киев, 1999.
6. Малыхина М.П. Базы данных: основы, проектирование, использование : учебное пособие для вузов по направлению подготовки «Информатика и вычислительная техника» / М.П. Малыхина. – СПб., 2006.
7. Марков А.С. Базы данных. Введение в теорию и методологию: учебник по специальности «Прикладная математика и информатика» / А.С. Марков, К.Ю. Лисовский. – М., 2006.
8. Советов Б.Я. Базы данных. Теория и практика / Б.Я. Советов, В.В. Цехановский, В.Д. Чертовской. – М., 2007.
9. Кириллов В.В. Введение в реляционные базы данных / В.В. Кириллов, Г. Громов. – СПб., 2009.
10. Грабер М. Введение в SQL / М. Грабер. – М.: ЛОРИ, 1996.
11. Уорсли Дж. PostgreSQL / Дж. Уорсли, Дж. Дрейк. – СПб.: Питер, 2003.
12. Сигнор Р. Использование ODBC для доступа к базам данных / Р. Сигнор, М.О. Стегман. – М.: Бином, 1995.
13. Архангельский А.Я. Программирование в C++Builder 6 / А.Я. Архангельский. – М.: БИНОМ, 2003.
14. Хьюгс Д. PHP. Руководство разработчика / Д. Хьюгс. – М.: Диасофт, 2001.
15. Дунаев В. Web-программирование для всех / В. Дунаев. – М.: БХВ-Петербург, 2012.
16. Мэт Зандстра. PHP. Объекты, шаблоны и методики программирования / Мэт Зандстра. – М.: Вильямс, 2011.

ОБЛАСТЬ СВЯЗИ SQL

```

struct sqlca_s
{
    long sqlcode;           /* код завершения: */
    char sqlerrm[72];       /* параметры сообщений об ошибке */
    char sqlerrp[8];        /* для внутреннего пользования */
    long sqlerrd[6];
        /* 0 – ожидаемое количество возвращаемых строк */
        /* 1 – значение поля serial после insert или ISAM-код
                               ошибки */
        /* 2 – количество обработанных строк */
        /* 3 – оценочное число обращений к диску */
        /* 4 – смещение ошибки в SQL-описании */
        /* 5 – rowid строки после вставки, удаления, коррек-
ровки */
    struct sqlcaw_s
    {
        char sqlwarn0; /* =W в случае любого sqlwarn[1–7] */
        char sqlwarn1; /* =W в случае любых усечений */
        char sqlwarn2; /* =W в случае возвращения пустого
значения */
        char sqlwarn3; /* =W, если список Select не совпадает
                               со списком полей */
        char sqlwarn4; /* =W, если нет where в операторах
                               Insert или Delete */
        char sqlwarn5; /* =W, если не ANSI-описание */
        char sqlwarn6; /* =W зарезервировано */
        char sqlwarn7; /* =W зарезервировано */
    } sqlwarn;
};

extern struct sqlca_s sqlca;
extern long SQLCODE;
#define SQLNOTFOUND 100;

```

КОДЫ ВОЗВРАТА SQLSTATE

01000 *Основное предупреждение*

Специфичное для драйвера информационное сообщение. Функция может возвращать `SQL_SUCCESS_INFO`. Этот код возврата указывает, что несмотря на успешное завершение функции, можно получить предупреждение или дополнительную информацию, вызывая `SQLError`.

01002 *Ошибка отсоединения*

Несмотря на ошибку во время отсоединения, отсоединение выполняется. Функция может возвращать `SQL_SUCCESS_WITH_INFO`. Этот код возврата указывает на то, что несмотря на успешное завершение функции, можно получить предупреждение или дополнительную информацию, вызывая `SQLError`.

01004 *Усечение данных*

Буфер был недостаточным для хранения всей возвращенной информации. Часто функции включают аргумент, содержащий длину данных, которые были усечены. Функция может возвращать `SQL_SUCCESS_WITH_INFO`. Этот код возврата указывает на то, что несмотря на успешное завершение функции, можно получить предупреждение или дополнительную информацию, вызывая `SQLError()`.

01S00 *Неправильная строка с атрибутами соединения*

Ключевое слово атрибута соединения является неправильным (функция возвращает `SQL_NEED_DATA`) или недопустимым для текущего уровня соединения. При вызове `SQLDriverConnect()` с неправильным ключевым атрибутом соединения было установлено соединение с источником данных. Функция может возвращать `SQL_SUCCESS_WITH_INFO`. Этот код возврата указывает на то, что несмотря на успешное завершение функции, можно получить предупреждение или дополнительную информацию, вызывая `SQLError()`.

01S01 *Ошибка в строке*

Ошибка возникла при извлечении одной или нескольких строк или при перемещении указателя строки на позицию для выполнения дейст-

вий, таких как добавление, удаление или модификация данных. Функция может возвращать `SQL_SUCCESS_WITH_INFO`. Этот код возврата указывает на то, что несмотря на успешное завершение функции, можно получить предупреждение или дополнительную информацию, вызывая **`SQLError()`**.

01S03 Не удалось удалить или модифицировать строки

Аргумент функции содержит позиционируемое удаление или модификацию, но не удалось удалить или модифицировать ни одну строку. Функция может возвращать `SQL_SUCCESS_WITH_INFO`. Этот код возврата указывает на то, что несмотря на успешное завершение функции, можно получить предупреждение или дополнительную информацию, вызывая **`SQLError()`**.

01S04 Были удалены или модифицируемы несколько строк

Аргумент функции содержит оператор, выполнение которого вызвало удаление или модификацию более одной строки. Функция может возвращать `SQL_SUCCESS_WITH_INFO`. Этот код возврата указывает на то, что несмотря на успешное завершение функции, можно получить предупреждение или дополнительную информацию, вызывая **`SQLError()`**.

07001 Неправильное количество параметров

Количество параметров в выполняемом SQL-операторе не совпадает с количеством параметров, заданным в **`SQLBindParameter()`**.

07006 Несовпадение типов данных

Значение не может быть преобразовано в указанный C-тип данных.

08001 Невозможно соединиться с источником данных

Драйвер не смог соединиться с источником данных.

08002 Соединение уже используется

Указанный идентификатор уже был использован для установления связи с источником данных, а соединение еще не закрыто.

08003 *Соединение не установлено*

Не удалось установить связь между идентификатором соединения и источником данных или необходимо выполнение операции, требующей соединения, которое к этому моменту не было установлено.

08004 *Источник данных разорвал установленную связь*

Источник данных разорвал установленную связь по внутренним причинам.

08S01 *Ошибка в связи с источником данных*

Связь между драйвером и источником данных, к которому драйвер пытается присоединиться, прервана до завершения выполнения функции.

22003 *Числовое значение вне допустимого диапазона*

В случае присвоения числового значения столбцу или переменной произошла потеря данных.

22005 *Ошибка присвоения*

Несовместимые типы данных параметра или значения столбцов связанной таблицы.

22008 *Неправильное значение даты*

Значение даты, переданное параметрам date, time, timestamp или столбцу, было неправильным.

23000 *Нарушение целостности данных*

SQL-оператор содержит параметр, который вызвал нарушение целостности данных. Например, значением параметра было NULL для столбца, определенного как NOT NULL, или имеет место дублирование значений для уникальных столбцов.

24000 *Неправильное состояние курсора*

Этот код ошибки возникает при следующих ситуациях.

1. Оператор, связанный с идентификатором оператора, не возвращает результирующее множество.

2. SQL-оператор содержит позиционируемые операторы UPDATE или DELETE, а курсор был позиционирован до начала или за результирующим множеством.

3. Идентификатор оператора в состоянии выполнения, но нет связанного с ним результирующего множества.

4. Для идентификатора оператора курсор был открыт, но не вызывались **SQLFetch()** или **SQLExtendedFetch()**.

5. Для идентификатора оператора было открыто результирующее множество, но не вызывались **SQLFetch()** или **SQLExtendedFetch()**.

42000 *Синтаксическая ошибка или ошибка доступа*

Пользователь не имеет достаточных полномочий для выполнения SQL-оператора, или же драйвер не может заблокировать таблицы для выполнения операции.

70100 *Операция прервана*

Источник данных не может обработать запрос на прекращение выполнения SQL-оператора.

IM002 *Не определены имя источника данных и драйвер по умолчанию*

1. Имя источника данных не найдено в файле `odbc.ini`.

2. Не найден файл `odbc.ini`.

S0001 *Таблица или представление данных уже существует*

SQL-оператор CREATE TABLE содержит имя существующей таблицы.

S0002 *Таблица не найдена*

SQL-оператор DROP TABLE содержит имя несуществующей таблицы.

S1009 *Неправильное значение аргумента*

Аргумент содержит неправильное значение.

S1010 *Ошибка последовательности вызовов*

1. Функции вызваны в недопустимой последовательности.

2. Вызвана асинхронно выполняющаяся функция, и в это время вызвана другая функция с таким же идентификатором оператора.

3. Были вызваны функции **SQLExecute()**, **SQLExecDirect()** и было возвращено значение **SQL_NEED_DATA**. Функция, отличная от указанных, вызвана до того, как были переданы все данные.

4. Указанный идентификатор оператора уже находится в подготовленном или выполняемом состоянии, и функция, вызывающая ошибку, должна вызываться до вызова **SQLPrepare()** или **SQLExecDirect()**.

5. Была вызвана функция, которая требует, чтобы идентификатор оператора находился в выполняемом состоянии, т. е. сначала должна вызываться функция **SQLExecute()** или **SQLExecDirect()**.

S1106 Недопустимый тип перемещения по результирующему множеству

Недопустимое значение аргумента **fFetchType** функции **SQLExtendedFetch()**. Например, значение **SQL_CURSOR_TYPE** было равно **SQL_CURSOR_FORWARD_ONLY**, а значение аргумента **fFetchType** не было равно **SQL_FETCH_NEXT**.

S1109 Неправильная позиция курсора

1. При выполнении позиционируемых операторов текущее положение курсора установлено на строку, которая имеет состояние **SQL_ROW_DELETED** или **SQL_ROW_ERROR**.

2. Курсор, связанный с идентификатором оператора, определен как перемещаемый только в прямом направлении, что препятствует позиционированию курсора внутри результирующего множества.

КОДЫ ВОЗВРАТА ПРОТОКОЛА HTML

Код возврата	Значение
200	Успешное выполнение
201	Успешная команда POST
202	Запрос принят
203	Запрос GET или HEAD выполнен
204	Запрос выполнен, но нет содержимого
300	Ресурс обнаружен в нескольких местах
301	Ресурс удален навсегда
302	Ресурс отсутствует временно
304	Ресурс был изменен
400	Плохой запрос от клиента
401	Неавторизованный запрос
402	Необходима оплата за ресурс
403	Доступ запрещен
404	Ресурс не найден
405	Метод не применим для этого вида ресурса
406	Недопустимый тип ресурса
410	Ресурс недоступен
500	Внутренняя ошибка сервера
501	Метод не выполнен
502	Неисправный шлюз либо перегрузка сервера
503	Сервер недоступен или тайм-аут шлюза
504	Вторичный шлюз или тайм-аут сервера