**Московский государственный
университет им. М.В. Ломоносова**

Факультет вычислительной математики и кибернетики

Волкова И.А., Головин И.Г., Мальковский М.Г.

Модельный SQL-интерпретатор

(Издание третье, переработанное)

2003

УДК 519.6+681.3.06

В данном методическом пособии описывается задание практикума на ЭВМ для студентов 2 курса факультета вычислительной математики и кибернетики в поддержку основного курса «Системное программное обеспечение». Приводятся подробные методические пояснения и рекомендации. Во втором издании частично изменена постановка задачи и исправлены замеченные неточности.

Составители пособия благодарят Богачева Д.А. за предоставленные материалы и Родина В.И. за ценные советы и замечания.

Рецензенты:
проф. Жоголев Е.А.
доц. Корухова Л.С.

Волкова И.А., Головин И.Г., Мальковский М.Г. «Модельный SQL-интерпретатор (Методическое пособие)» — издание третье, переработанное.

Издательский отдел факультета ВМиК МГУ
(лицензия ЛР №040777 от 23.07.96), 2003.— 30 с.

Печатается по решению Редакционно-Издательского Совета факультета вычислительной математики и кибернетики МГУ им. М.В. Ломоносова.

ISBN 5-89407-033-3

ПОСТАНОВКА ЗАДАЧИ

Задача реализации модельного SQL-интерпретатора разбивается на следующие подзадачи:

1. Реализация архитектуры «Клиент — Сервер».
2. Реализация SQL-интерпретатора.
3. Реализация модельного SQL-сервера на базе разработанного интерпретатора и предоставленной библиотеки классов для работы с файлами данных.
4. Реализация интерфейса пользователя с модельным SQL-сервером.
5. Реализация модельной базы данных, демонстрирующей работоспособность разработанной программы.

Язык реализации - C++.

Архитектура «Клиент — Сервер» предполагает наличие одного или нескольких (в зависимости от варианта задания) процессов-клиентов, принимающих запрос к реляционной базе данных, записанный на модельном SQL. Клиентский процесс передает запрос процессу-серверу, который осуществляет поиск в базе данных в соответствии с запросом и передает результат поиска клиенту.

SQL-интерпретатор реализует некоторое подмножество предложений языка SQL, называемое далее модельным SQL. Описание модельного SQL см. ниже в разделе «Методические указания». Синтаксический анализ SQL-предложений рекомендуется проводить методом рекурсивного спуска [4]. Интерпретацию SQL-предложений необходимо реализовать с помощью библиотеки классов для работы с файлами данных, описание которой приведено в п. 2.

Интерфейс пользователя должен давать возможность вводить и редактировать SQL-запросы и просматривать их результаты в табличном виде. Конкретные детали интерфейса оставляются на усмотрение преподавателя.

ВАРИАНТЫ ЗАДАНИЯ

I. Архитектура «Клиент — Сервер».

1. Один клиент. Клиент и сервер на одной ЭВМ.
2. Один клиент. Клиент и сервер в сети ЭВМ.
3. Несколько клиентов. Клиенты и сервер на одной ЭВМ.
4. Несколько клиентов. Клиенты и сервер в сети ЭВМ.

II. Функции и способ взаимодействия клиента и сервера.

1. Клиент получает от пользователя запрос на модельном SQL, не анализируя его, передает серверу. Сервер анализирует запрос и в случае его корректности выполняет запрос и передает клиенту ответ. Если же запрос некорректен, сервер передает клиенту соответствующий код ошибки. Клиент выдает пользователю либо ответ на его запрос, либо сообщение об ошибке в каком-либо удобном виде.
2. Клиент получает от пользователя запрос на SQL, анализирует его и в случае ошибки сообщает об этом пользователю, иначе передает серверу запрос в некотором внутреннем представлении. Сервер обращается к БД, определяет ответ на запрос и передает

его клиенту. Клиент выдает пользователю ответ сервера в каком-либо удобном виде.

ТРЕБОВАНИЯ К РЕАЛИЗАЦИИ

Необходимо реализовать модельный SQL-интерпретатор в объектно-ориентированном стиле с использованием дополнительных возможностей языка реализации C++ по сравнению, например, с Си:

- разработать архитектуру программы,
- продумать и зафиксировать содержание и иерархию вводимых классов,
- использовать встроенный в C++ аппарат обработки исключительных ситуаций в программе,
- при необходимости использовать стандартные библиотеки C++.

СОДЕРЖАНИЕ ОТЧЕТА

1. Постановка задачи (конкретный вариант).
2. Описание способа взаимодействия процесса-клиента и процесса-сервера.
3. Описание архитектуры программы и основных классов.
4. Описание интерфейса пользователя с процессом-клиентом, перечень типов обнаруживаемых ошибок и выдаваемых диагностических сообщений.
5. Листинг основных программ и классов на языке Си++ (по просьбе преподавателя).

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

1. Моделирование архитектуры «Клиент — Сервер».

Межпроцессное взаимодействие можно организовать, используя модель «Клиент — Сервер». В этой модели один процесс, называемый сервером, отвечает за обработку запросов, получаемых им от других процессов — клиентов. Таким образом, клиент-серверная архитектура тесно связана с механизмами межпроцессного взаимодействия. Традиционный механизм информационных каналов (pipe) не подходит для клиент-серверных приложений, так как каналы могут связывать только процессы, запущенные одним пользователем. Кроме того, клиентские и серверные процессы могут выполняться на разных компьютерах (и даже в разных операционных системах), что предъявляет особые требования к механизму межпроцессной связи. Таким требованиям удовлетворяет механизм сокетов, кратко описываемый ниже.

Обычно, когда вызывается программа сервер, она запрашивает у операционной системы сокет (socket) (средство для соединения, связи процессов). Когда сервер получает сокет, он связывает с ним некоторое стандартное имя, чтобы программы-клиенты могли общаться с сервером через данный сокет по этому имени.

После присваивания имени сокету, сервер «слушает» на этом сокете требования связи (запросы) от процессов-клиентов. Когда запрос появляется, сервер может допустить или запретить связь клиента с сервером. Если связь допустима, ОС соединяет клиента с сервером, и сервер получает возможность получать сообщения от клиента и посылать ему ответы аналогично механизму информационных каналов.

Клиент также запрашивает у ОС свой сокет для взаимодействия с другим процессом (сервером), а затем сообщает имя сокета, с которым он хотел бы связаться. ОС пытается найти сокет с заданным именем и, если находит его, посылает серверу, слушающему этот сокет, запрос связи. Если сервер допускает связь, ОС создает специальный сокет, соединяющий два процесса, и клиент получает возможность посылать и получать данные от сервера аналогично механизму информационных каналов.

Средства межпроцессного взаимодействия для одной ЭВМ (ОС Berkeley UNIX)

Функция socket.

Эта функция используется для создания сокета.

Прототип:

```
int socket (int domain, int type, int protocol);
```

Первый параметр — домен — накладывает определенные ограничения на формат используемых процессом адресов и их интерпретацию. При работе с одной ЭВМ используется UNIX-домен, где адреса интерпретируются как имена файлов в UNIX. В этом случае в качестве первого параметра указывается константа AF_UNIX (AF — **A**ddress **F**amily).

Второй параметр определяет тип канала связи с сокетом, который должен быть использован.

Существует несколько типов каналов связи с сокетом, доступных при межпроцессном взаимодействии в UNIX, но обычно используются следующие два:

SOCK_STREAM — при этом типе связи поступающим в канал байтам информации гарантируется «доставка» в порядке их поступления; пока непрерывный поток байтов не прекратится, никакие другие данные приниматься каналом не будут (аналогом такой связи является механизм информационных каналов);

SOCK_DGRAM — этот тип связи используется для посылки отдельных пакетов информации, называемых дейтаграммами (сообщениями); при этом не гарантируется, что пакеты будут доставлены на место назначения в порядке поступления, а в действительности не гарантируется, что они все вообще будут доставлены (пример такого типа связи — доставка незаказного письма через обычную почту).

Третий параметр позволяет программисту выбрать нужный протокол для канала связи. Если этот параметр равен нулю, ОС выберет нужный протокол автоматически.

Функция socket возвращает целое положительное число — номер сокет-дескриптора (который можно использовать, например, в функциях read и write аналогично файловому дескриптору). Если же сокет по каким-либо причинам не был создан (например, очень много открытых файлов), возвращается -1, а в переменную errno записывается причина неудачи.

Константы, используемые в качестве аргументов при вызове socket, определены во включаемых файлах sys/socket.h и sys/types.h.

Функция bind.

Эта функция используется сервером для присваивания сокету имени. До выполнения функции bind (т.е. присваивания какого-либо имени, вид которого зависит от адресного домена) сокет недоступен программам-клиентам.

Прототип:

```
int bind(int s, struct sockaddr * name, int namelen);
```

Первый параметр — сокет-дескриптор, который данная функция именует. Второй параметр — указатель на структуру имени сокета **struct** sockaddr**.** Тип этой структуры зависит от домена. Для UNIX-домена этот тип называется sockaddr_un, он определен во включаемом файле sys/un.h и выглядит таким образом:

```
struct sockaddr_un {
   short sun_family;
   char  sun_path[108];
};
```

В качестве первого элемента структуры, обозначающего класс адресов, мы будем использовать константу AF_UNIX, второй элемент — имя файла, который будет соответствовать используемому сокету.

Файл с именем, указанным в sun_path, действительно создается, поэтому после окончания работы с данным сокетом надо выполнить функцию unlink, в противном случае другие программы, которые попытаются использовать данное имя, получат сообщение об ошибке.

Функция listen.

Функция listen используется сервером, чтобы информировать ОС, что он ожидает («слушает») запросы связи на данном сокете. Без такой функции всякое требование связи с этим сокетом будет отвергнуто.

Прототип:

```
int listen(int s, int backlog);
```

Первый аргумент — сокет для прослушивания, второй аргумент (backlog) — целое положительное число, определяющее, как много запросов связи может быть принято на сокет одновременно. В большинстве систем это значение должно быть не больше пяти. Заметим, что это число не имеет отношения к числу соединений, которое может поддерживаться сервером. Аргумент backlog имеет отношение только к числу запросов на соединение, которые приходят одновременно. Число установленных соединений может превышать это число.

Функция accept.

Эта функция используется сервером для принятия связи на сокет. Сокет должен быть уже слушающим в момент вызова функции. Если сервер устанавливает связь с клиентом, то функция accept возвращает новый сокет-дескриптор, через который и происходит общение клиента с сервером. Пока устанавливается связь клиента с сервером, функция accept блокирует другие запросы связи с данным сервером, а после установления связи «прослушивание» запросов возобновляется.

Прототип:

```
int accept(int s, struct sockaddr * name, int * anamelen);
```

Первый аргумент функции — сокет-дескриптор для принятия связей от клиентов. Второй аргумент — указатель на адрес клиента (структура sockaddr) для соответствующего домена. Третий аргумент — указатель на целое число — длину структуры адреса. Второй и третий аргументы заполняются

соответствующими значениями в момент установления связи клиента с сервером и позволяют серверу точно определить, с каким именно клиентом он общается. Если сервер не интересуется адресом клиента, в качестве второго и третьего аргументов можно задать NULL-указатели.

Функция connect.

Функция connect используется процессом-клиентом для установления связи с сервером.

Прототип:

```
int connect(int s, struct sockaddr * name, int namelen);
```

Первый аргумент — сокет-дескриптор клиента. Второй аргумент — указатель на адрес сервера (структура sockaddr) для соответствующего домена. Третий аргумент — целое число — длина структуры адреса.

Функция возвращает 0, если вызов успешный, и -1 иначе.

Функция send.

Функция служит для записи данных в сокет.

Прототип:

```
int send(int s, void * buf, int len, int flags);
```

Первый аргумент — сокет-дескриптор, в который записываются данные. Второй и третий аргументы — соответственно, адрес и длина буфера с записываемыми данными. Четвертый параметр — это комбинация битовых флагов, управляющих режимами записи. Если аргумент flags равен нулю, то запись в сокет (и, соответственно, считывание) происходит в порядке поступления байтов. Если значение flags есть MSG_OOB, то записываемые данные передаются потребителю вне очереди.

Функция возвращает число записанных в сокет байтов (в нормальном случае должно быть равно значению параметра len) или -1 в случае ошибки. Отметим, что запись в сокет не означает, что данные приняты на другом конце соединения процессом-потребителем. Для этого процесс-потребитель должен выполнить функцию recv (см. ниже). Таким образом, функции чтения и записи в сокет выполняются асинхронно.

Функция recv.

Функция служит для чтения данных из сокета.

Прототип:

```
int recv(int s, void * buf, int len, int flags);
```

Первый аргумент — сокет-дескриптор, из которого читаются данные. Второй и третий аргументы — соответственно, адрес и длина буфера для записи читаемых данных. Четвертый параметр — это комбинация битовых флагов, управляющих режимами чтения. Если аргумент flags равен нулю, то считанные данные удаляются из сокета. Если значение flags есть MSG_PEEK, то данные не удаляются и могут быть считаны последующим вызовом (или вызовами) recv.

Функция возвращает число считанных байтов или -1 в случае ошибки. Следует отметить, что нулевое значение не является ошибкой. Оно

сигнализирует об отсутствии записанных в сокет процессом-поставщиком данных.

Функция shutdown.

Эта функция используется для немедленного закрытия всех или части связей на сокет.

Прототип:

```
int shutdown(int s, int how);
```

Первый аргумент функции — сокет-дескриптор, который должен быть закрыт. Второй аргумент — целое значение, указывающее, каким образом закрывается сокет, а именно:

RD — сокет закрывается для чтения;
WR — сокет закрывается для записи;
RDWR — сокет закрывается для чтения и для записи.

Функция close.

Эта функция закрывает сокет и разрывает все соединения с этим сокетом. В отличие от функции shutdown функция close может дожидаться окончания всех операций с сокетом, обеспечивая «нормальное», а не аварийное закрытие соединений.

Прототип:

```
int close (int s);
```

Аргумент функции — закрываемый сокет-дескриптор.

Пример-оболочка программы “Клиент”

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <stdio.h>
#define ADDRESS “mysocket” /* адрес для связи */
void main ()
{  char c;
   int i, s, len;
   FILE *fp;
   struct sockaddr_un sa;
/* получаем свой сокет-дескриптор: */
   if ((s = socket (AF_UNIX, SOCK_STREAM, 0))<0){         perror
(“client: socket”); exit (1);
   }
/*  создаем адрес, по которому будем связываться с сервером: */
   sa.sun_family = AF_UNIX;
   strcpy (sa.sun_path, ADDRESS);
/* пытаемся связаться с сервером: */
   len = sizeof ( sa.sun_family) + strlen ( sa.sun_path);
   if ( connect ( s, &sa, len) < 0 ){
         perror (“client: connect”); exit (1);
   }

/*------------------------------------------- */
```

```
/* читаем сообщения сервера, пишем серверу: */
   fp = fdopen (s, "r");
   c = fgetc (fp);
/* ........................... */
   send (s, "client", 7, 0);
/* ........................... */
   close (s);
   exit (0);
}
```

Пример-оболочка программы “Сервер”

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <stdio.h>
#define ADDRESS "mysocket" /* адрес для связи */
void main ()
{  char c;
   int i, d, d1, len, ca_len;
   FILE *fp;
   struct sockaddr_un sa, ca;
/* получаем свой сокет-дескриптор: */
   if((d = socket (AF_UNIX, SOCK_STREAM, 0)) < 0){
         perror ("client: socket"); exit (1);
   }
/*  создаем адрес, с которым будут связываться клиенты */
   sa.sun_family = AF_UNIX;
   strcpy (sa.sun_path, ADDRESS);
/*  связываем адрес с сокетом;
   уничтожаем файл с именем ADDRESS, если он существует,
   для того, чтобы вызов bind завершился успешно
*/
   unlink (ADDRESS);
   len = sizeof ( sa.sun_family) + strlen (sa.sun_path);
   if ( bind ( d, &sa, len) < 0 ) {
         perror ("server: bind"); exit (1);
   }
/*  слушаем запросы на сокет */
   if ( listen ( d, 5) < 0 ) {
         perror ("server: listen"); exit (1);
   }
/* связываемся с клиентом через неименованный сокет с
   дескриптором d1:
*/
   if (( d1 = accept ( d, &ca, &ca_len)) < 0 ) {
         perror ("server: accept"); exit (1);
   }

/* ---------------------------------------- */

/* читаем запросы клиента, пишем клиенту: */
   fp = fdopen (d1, "r");
   c = fgetc (fp);
/* ............................. */
   send (d1, "server", 7, 0);
```

```
/* ............................... */
   putchar ( c );
/* ............................... */
   close (d1);
   exit (0);
}
```

Далее в качестве примеров приводятся оболочки программ для «Клиента» и «Сервера», основанных на объектно-ориентированном интерфейсе для работы с сокетами (одна из возможных реализаций интерфейса приводится в приложении).

Пример-оболочка программы «Клиент»
в объектно-ориентированном стиле

```
#include <iostream>
#include "sock_wrap.h" //см. приложение

using namespace std;
using namespace ModelSQL;

const char* address = "mysocket" // имя сокета

int main(int argc, char* argv[])
{
    try {
    // создаём сокет
    UnClientSocket sock( address );

    // устанавливаем соединение
    sock.Connect();

    // отправляем серверу строку
    sock.PutString("Hello from client!");

    // печатаем на экран ответ от сервера
    cout << "Read from server: " << sock.GetString() << endl;

    // продолжаем диалог с сервером, пока в этом есть необходимость
    // ...

    } catch (Exception& e) {
    // ошибка --- выводим текст сообщения на экран
    e.Report();
    }
    return 0;
}
```

Пример-оболочка программы «Сервер»
в объектно-ориентированном стиле

```
#include <iostream>
#include "sock_wrap.h" //см. приложение
```

```
using namespace std;
using namespace ModelSQL;

const char* address = "mysocket" // имя сокета

class MyServerSocket : public UnServerSocket {
public:
    MyServerSocket () : UnServerSocket (address) {}
protected:
    void OnAccept (BaseSocket * pConn)
    {
    // установлено соединение с клиентом, читаем сообщение
    cout << "Read from client: " << pConn->GetString() << endl;

    // отправляем ответ
    pConn->PutString("Hello from server.");

    // продолжаем диалог с клиентом, пока в этом есть необходимость
    // ...

    delete pConn;
    }
};

int main(int argc, char* argv[])
{
    try {
      // создаём серверный сокет
      MyServerSocket sock;

      for (;;)
        // слушаем запросы на соединение
        sock.Accept();
    } catch (Exception& e) {
      // ошибка --- выводим текст сообщения на экран
        e.Report();
    }
    return 0;
}
```

Средства межпроцессного взаимодействия для сети ЭВМ (ОС Berkeley UNIX)

Сетевые средства ОС UNIX также базируются на механизме сокетов, рассмотренном выше, но работают в Internet-домене.

В UNIX домене сетевой адрес процесса специфицируется стандартным описанием пути к некоторому файлу. Этот домен не подходит для работы с сетью, так как не все компьютеры, связанные в сеть, работают в ОС UNIX, а следовательно, могут иметь другие способы спецификации адресов, поэтому

при организации межпроцессного взаимодействия для сети ЭВМ используется Internet-домен.

Адресация в Internet-домене происходит следующим образом. Каждый компьютер в сети имеет символическое сетевое имя (hostname) и уникальный 32-битный сетевой адрес (IP-адрес). Для установления сетевого соединения с конкретным процессом, запущенном на некотором компьютере в сети, необходимо также указать 16-битный номер порта, через который данный процесс готов установить сетевое соединение (пользовательским программам могут выдаваться порты с номерами от 1025 до 32767). Некоторые системные обслуживающие процессы имеют закреплённые стандартные номера портов, такие процессы называют сетевыми сервисами. Так, например, сервис передачи файлов по протоколу FTP использует порт 21. Поэтому программа, желающая связаться с FTP-сервером на компьютере с сетевым адресом 12345, должна обратиться по Internet-адресу (12345, 21).

Преобразование сетевого имени в сетевой адрес

Пользователю обычно известно сетевое имя своего компьютера. Если же это не так, то можно воспользоваться функцией gethostname.

Функция gethostname.

Прототип:

```
int gethostname (char * buffer, int buflen);
```

Первый аргумент функции — символьный буфер, в который в результате выполнения функции запишется строка — сетевое имя компьютера; второй аргумент — целое число — длина символьного буфера. В случае если длины буфера не хватает для хранения сетевого имени, функция возвращает -1, иначе — 0.

Чтобы использовать сетевое имя в качестве сетевого адреса, надо его преобразовать. Соответствия между сетевыми именами и сетевыми адресами хранятся в системном текстовом файле /etc/hosts.

Функция gethostbyname.

Прототип:

```
struct hostent * gethostbyname (char * hostname);
```

У этой функции один аргумент (строка символов hostname) — сетевое имя. Она возвращает указатель на структуру **struct** hostent, определенную во включаемом файле <netdb.h>:

```
struct hostent {
   char*  h_name;           /* hostname ЭВМ */
   char** h_aliases;  /* список синонимов */
   int    h_addrtype; /* тип адресов ЭВМ */
   int    h_length;         /* длина адреса */
   char** h_addr_list /* список адресов (для разных
сетей) */
#define  h_addr  h_addr_list[0]
};
```

Если же указанного сетевого имени нет в базе данных, возвращается константа NULL.

Получение номера порта

Для всех стандартных сетевых сервисов существует перечень закреплённых за ними номеров портов в текстовом файле /etc/services. Это облегчает программе-клиенту производить соединение со стандартным сетевым сервисом на любой другой машине.

Функция getservbyname.

Эта функция используется для того, чтобы получить номер порта конкретной обслуживающей программы.

Прототип:

```
struct servent* getservbyname(char * name, char * proto);
```

Первый аргумент функции — символическое имя стандартного сервиса ("telnet", "ftp", "smtp", "pop3" и т.п.), второй аргумент — символическое имя транспортного протокола: "tcp" или "udp". Второй аргумент нужен для того, чтобы определить, нужно ли устанавливать сессию для гарантированной доставки пакетов с сохранением очерёдности SOCK_STREAM ("tcp") или достаточно передачи коротких сообщений SOCK_DGRAM ("udp").

Функция возвращает указатель на структуру типа **struct** servent, определенную во включаемом файле <netdb.h>:

```
struct servent {
char * s_name;     /* имя обслуживающей программы */
char** s_aliases; /* список синонимов */
int    s_port;    /* номер порта */
char*  s_proto;   /* используемый протокол */
};
```

Если имя сервиса не найдено в базе данных, возвращается константа NULL.

Обычные пользовательские программы могут использовать любые свободные порты, но их номера должны лежать в пределах от 1025 до 32767 в соответствии с требованиями ОС UNIX и ARPANET (international network administration), зарезервировавших номера меньшие 1025 для стандартных и системных сервисов.

Порядок байтов в сети

Известно, что целые числа хранятся в различных компьютерах разными способами. Некоторые компьютеры хранят целые числа в перевернутом виде, т.е. старшие байты числа имеют меньший адрес по сравнению с младшим байтом, в других же наоборот старший байт имеет больший адрес. Чтобы избежать путаницы при взаимодействии машин, использующих различный порядок байтов, *network software* требует, чтобы все целочисленные данные были представлены в сетевом байтовом порядке. Для того чтобы привести целые числа к сетевому байтовому порядку, используются следующие две функции.

Функция htons.

Прототип:

```
short htons (short i);
```

Функция htonl.

Прототип:

```
long htonl (long i);
```

Эти функции переводят короткие (htons) и длинные (htonl) целые числа из байтового порядка компьютера в сетевой.

Следующие две функции производят обратные действия, т.е. переводят короткие и длинные целые числа соответственно из сетевого байтового порядка в представление, принятое на данном компьютере.

Функция ntohs.

Прототип:

```
short ntohs (short i);
```

Функция ntohl.

Прототип:

```
long ntohl (long i);
```

Замечание: функции gethostbyname и getservbyname возвращают все данные и их структуры в сетевом байтовом порядке, т.е. дополнительные преобразования не требуются.

Функции работы с сетью ЭВМ

Системные функции для межпроцессного взаимодействия, используемые при решении сетевых задач, те же, что и для межпроцессного взаимодействия на одной машине. Существует лишь несколько различий в параметрах, передаваемых этим системным функциям.

Во-первых, в качестве первого параметра функции socket используется константа AF_INET, определяющая Internet-домен. В качестве второго параметра также указывается либо SOCK_STREAM либо SOCK_DGRAM.

Во вторых, в качестве типа sockaddr-структуры, используемой функциями accept, bind, connect, send и recv, употребляется тип sockadd_in, который объявлен во включаемом файле <netinet/in.h>:

```
struct sockaddr_in   {
   short       sin_family;
   u_short     sin_port;   /* номер порта */
   struct in_addr    sin_addr;   /* сетевой адрес ЭВМ */
   char        sin_zero[8];
};
struct in_addr {
   u_long s_addr;
};
```

Типы u_short (**unsigned short**) и u_long (**unsigned long**) определены во включаемом файле <sys/types.h>. Номера портов и сетевые номера компьютеров должны указываться в сетевом байтовом порядке.

Пример-оболочка программы “Клиент” для сети ЭВМ

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
extern int errno;
void main ()
{  char c;
   int s;
   FILE *fp;
   char hostname[64];
   struct hostent *hp;
   struct sockaddr_in sin;
/* в этом примере клиент и сервер выполняются на одном компьютере,
но программа легко обобщается на случай разных компьютеров. Для
этого можно, например, использовать хост-имя не собственного
компьютера, как ниже, а имя компьютера, на котором выполняется
процесс-сервер */
/* прежде всего получаем hostname собственной ЭВМ: */
   gethostname (hostname, sizeof (hostname));
/* затем определяем сетевой номер своей машины:    */
   if ((hp = gethostbyname (hostname)) == NULL){
         fprintf (stderr, "%s: unknown host.\n",
                                          hostname);
         exit (1);
}
/* получаем свой сокет-дескриптор: */
   if ((s = socket (AF_INET, SOCK_STREAM, 0)) < 0) {
   perror ("client: socket"); exit (1);
   }
/* создаем адрес, по которому будем связываться с сервером: */
   sin.sin_family = AF_INET;
   sin.sin_port = htons (1234);
/* копируем сетевой номер: */
   bcopy (hp -> h_addr, &sin.sin_addr, hp ->
                            h_length);
/* пытаемся связаться с сервером: */
   if ( connect ( s, &sin, sizeof (sin)) < 0 ) {
         perror ("client: connect"); exit (1);
   }
/* ---------------------------------------- */

/* читаем сообщения сервера, пишем серверу: */
   fp = fdopen (s, "r");
   c = fgetc (fp);
/* ............................... */
   send (s, "client", 7, 0);
/* ............................... */
   close (s);
   exit (0);
}
```

Пример-оболочка программы “Сервер” для сети ЭВМ

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
extern int errno;
void main (){
   char c;
   int d, d1, fromlen;
   FILE *fp;
   char hostname[64];
   struct hostent *hp;
   struct sockaddr_in sin, fromsin;
/* получаем хост-имя собственной ЭВМ: */
   gethostname (hostname, sizeof (hostname));
/* определяем сетевой номер своей машины: */
   if ((hp = gethostbyname (hostname)) == NULL) {
         fprintf (stderr, "%s: unknown host.\n", hostname);
         exit (1);
   }
/* получаем свой сокет-дескриптор: */
   if ((d = socket (AF_INET, SOCK_STREAM, 0)) < 0) {
   perror ("client: socket"); exit (1);
   }
/* создаем адрес, который свяжется с сокетом: */
   sin.sin_family = AF_INET;
   sin.sin_port = htons (1234);
/*       копируем сетевой номер: */
   bcopy (hp->h_addr, &sin.sin_addr, hp->h_length);
/*  связываем адрес с сокетом */
   if ( bind ( d, &sin, sizeof (sin)) < 0 ){
         perror ("server: bind"); exit (1);
   }
/*  слушаем запросы на сокет */
   if ( listen ( d, 5) < 0 ) {
         perror ("server: listen"); exit (1);
   }
/* связываемся с клиентом через неименованный сокет с дескриптором
d1; после установления связи адрес клиента содержится в структуре
fromsin */

   if (( d1 = accept ( d, &fromsin, &fromlen)) < 0 )      {
   perror ("server: accept"); exit (1);
   }
/* ---------------------------------------- */
/* читаем сообщения клиента, пишем клиенту: */
   fp = fdopen (d1, "r");
   c = fgetc (fp);
/* ................................. */
   send (d1, "server", 7, 0);
/* ................................. */
   putchar (c);
```

```
/* ................................. */
   close (d);
   exit (0);
}
```

Ниже приводятся примеры оболочек программ для «Клиента» и «Сервера» для сети ЭВМ, основанных на объектно-ориентированном интерфейсе для работы с сокетами (одна из возможных реализаций интерфейса приводится в приложении).

Пример-оболочка программы «Клиент» для сети ЭВМ в объектно-ориентированном стиле

```
#include <iostream>
#include "sock_wrap.h" //см. приложение

using namespace std;
using namespace ModelSQL;

#define PORT_NUM 1234          // номер порта процесса-сервера

//В этом примере клиент и сервер выполняются на одном компьютере,
//но программа легко обобщается на случай разных компьютеров. Для
//этого можно, например, использовать сетевое имя не собственного
//компьютера, как ниже, а имя компьютера, на котором выполняется
//процесс-сервер
*/

int main(int argc, char* argv[])
{
    try {
      char host[64];

    // запрос сетевого имени собственной ЭВМ
    if (gethostname(host, sizeof host) < 0) {
        // ошибка --- досрочно завершаем выполнение
        cerr << GetLastError();
        perror("Host name");
        exit (-1);
      }

    // создаём сокет
    InClientSocket sock(host, PORT_NUM);

    // устанавливаем соединение
    sock.Connect();

    // отправляем серверу строку
    sock.PutString("Hello from client!");

    // печатаем на экран ответ от сервера
     cout << "Read from server: " << sock.GetString() << endl;

    // продолжаем диалог с сервером, пока в этом есть необходимость
    // ...
```

```
    } catch (Exception& e) {
        // ошибка --- выводим текст сообщения на экран
        e.Report();
    }
    return 0;
}
```

Пример-оболочка программы «Сервер» для сети ЭВМ в объектно-ориентированном стиле

```
#include <iostream>
#include "sock_wrap.h" //см. приложение

using namespace std;
using namespace ModelSQL;

#define PORT_NUM 1234  // номер порта процесса-сервера

class MyServerSocket : public InServerSocket {
public:
    MyServerSocket () : InServerSocket (PORT_NUM) {}
protected:
    void OnAccept (BaseSocket * pConn)
    {
    // установлено соединение с клиентом, читаем сообщение
    cout << "Read from client: " << pConn->GetString() << endl;
    // отправляем ответ
    pConn->PutString("Hello from server.");

    // продолжаем диалог с клиентом, пока в этом есть необходимость
    // ...

    delete pConn;
    }
};

int main(int argc, char* argv[])
{
    try {
      // создаём серверный сокет
      MyServerSocket sock;

      for (;;)
          // слушаем запросы на соединение
          sock.Accept();
    } catch (Exception& e) {
      // ошибка --- выводим текст сообщения на экран
      e.Report();
    }
    return 0;
}
```

2. БД и СУБД

Обычно, говоря о базе данных, подразумевают некоторую систему базы данных, имеющую в своем составе следующие основные компоненты: непосредственно базу данных (БД), систему управления базой данных (СУБД) и язык данных (язык общения пользователя с БД, в нашем случае это модельный язык SQL).

Собственно БД — совокупность данных, хранящихся во внешней памяти. В системе базы данных обычно выделяют три основных уровня хранения информации: внутренний, концептуальный и внешний. Внутренний уровень наиболее близок к физической памяти, т.е. связан со способом фактического хранения информации. Внешний уровень наиболее близок к пользователям, т.е. связан с тем, как отдельные пользователи представляют себе эти данные. Концептуальный уровень — промежуточный уровень между двумя другими — есть представление полного информационного содержания базы данных в несколько абстрактной форме по сравнению со способом физического хранения данных.

СУБД — программа, которая управляет всем доступом к базе данных. В общих чертах это происходит следующим образом:

— пользователь выдает запрос на доступ к БД, используя конкретный язык данных;

— СУБД принимает запрос, анализирует и интерпретирует его, обследуя по очереди внешнее представление, отображение “внешний - концепту-альный”, концептуальное представление, отображение “концептуальный - внутренний” и структуру хранения;

— СУБД выполняет необходимые операции над хранимой БД;

— СУБД выдает ответ пользователю, сначала выбирая все требуемыsse экземпляры хранимых записей, затем получая экземпляры концептуальных записей и уже потом формируя требуемый экземпляр внешней записи.

Ядром любой системы данных является модель данных, которая в частности определяет выбор языка данных. Имеются три широко известные модели данных: реляционная, иерархическая и сетевая [1].

В данной разработке для хранения информации предлагается создать реляционную БД, состоящую из нескольких (2-5) таблиц.

На логическом (концептуальном) и внешнем уровнях каждая таблица представляет собой совокупность записей. Все записи одной таблицы имеют одинаковую структуру. Каждая запись состоит из нескольких полей.

С каждым полем связываются следующие характеристики: имя поля, тип поля (домен), длина поля и значение поля.

Имена, типы, и длины полей задаются при определении структуры таблицы.

На физическом (внутреннем) уровне таблицы предлагается реализовать с помощью готовой библиотеки функций для работы с файлами данных. Файл с таблицей помимо данных (записей) содержит описание структуры таблицы, определяющей поля данной таблицы. Эта структура загружается в оперативную память при открытии соответствующей таблицы и используется при всех операциях, производимых над таблицами.

Для работы с таблицами реализована следующая функциональность:

— создание новой таблицы,

— удаление существующей таблицы,

— открытие существующей таблицы,
— перемещение по записям таблицы,
— редактирование отдельных записей таблицы,
— добавление новых записей в таблицу,
— удаление существующих записей из таблицы.

Операции работы с таблицами реализованы через операции базового (<io.h>) ввода-вывода:

— создание или открытие файла (open),
— закрытие файла (close),
— удаление файла (unlink),
— чтение из файла (read),
— запись в файл (write),
— установка на определенную позицию в файле (lseek).

Каждая таблица описывается одним файлом данных. Этот файл состоит из двух частей — заголовка, в котором описывается структура таблицы, и собственно записей. Заголовок содержит информацию о числе полей и для каждого поля — его тип (строковый или длинный целый). Для строковых полей, кроме того, хранится их длина.

При выполнении задания для работы с таблицами БД необходимо использовать предоставляемую преподавателями библиотеку (файлы MSQLTab.cpp, MSQLTab.h, table.c, table.h), объектно-ориентированный интерфейс которой описан ниже (файл MSQLTab.h).

```
#ifndef __MSQLTAB_H__
#define __MSQLTAB_H__

#include <string>
#include <vector>

namespace ModelSQL {

// EFieldType - описание типов полей таблицы
enum EFieldType {
   EFTText,
   EFTLong,
   EFTBool,
};

// SQLTableException - класс исключений, генерируемых при работе с
//таблицами
class SQLTableException : public Exception {
public:
   SQLTableException(Errors errcode) : Exception(errcode) {}
   char * GetMessage()
};

// EOpCode - определяет возможные коды ошибок при работе с
//таблицами
enum EOpCode {
   OpLike,
   OpNotLike,
   OpAdd,
```

```
  OpSub,
  OpNegate,
  OpMult,
  OpDiv,
  OpMod,
  OpOr,
  OpAnd,
  OpNot,
  OpEq,
  OpNeq,
  OpLess,
  OpGreater,
  OpLessEq,
  OpGreaterEq,
};

// SQLValue - базовый абстрактный класс для поля таблицы
class SQLValue {
public:
    virtual EFieldType GetType() = 0;
    virtual SQLValue * Execute(EOpCode Op, SQLValue * pRight) = 0;
    virtual void       GetValue(long& val) = 0;
    virtual void       GetValue(std::string& val) = 0;
};

typedef SQLValue * (*OpHandler)(SQLValue * pLeft, SQLValue *
pRight);

// SQLTextValue - представление текстового поля
class SQLTextValue : public SQLValue {
    std::string      str;
    static OpHandler Ops[];
public:
    SQLTextValue(const std::string& s) : str(s) {}
    EFieldType GetType();
    SQLValue * Execute(EOpCode Op, SQLValue * pRight);
    void       GetValue(long& val);
               { throw SQLTableException(IllegalType); }
    void       GetValue(std::string& val);
};

// SQLLongValue - представление целого поля
class SQLLongValue : public SQLValue {
    long l;
    static OpHandler Ops[];
public:
    SQLLongValue(long val) : l(val) {}
    EFieldType GetType();
    SQLValue * Execute(EOpCode Op, SQLValue * pRight);
    void       GetValue(long& val)
    void       GetValue(std::string& val)
               { throw SQLTableException(IllegalType); }
};

// TableRec - базовый класс записи (кортежа) таблицы
class TableRec {
public:
    TableRec(THandle& h) : tab(h) {}
```

```
    // Устанавливает новое значение поля
    virtual void SetFieldValue(const char * FieldName, SQLValue *
pValue) = 0;
    // Возвращает значение поля
    virtual SQLValue * GetFieldValue(const char * FieldName) = 0;
protected:
    THandle& tab;
};

// NewTableRec - представление новой записи
class NewTableRec : public TableRec {
public:
    NewTableRec(THandle tab) : TableRec(tab);
    void SetFieldValue(const char * FieldName, SQLValue * pValue);
    // cannot read new rec fields
    SQLValue * GetFieldValue(const char * FieldName)
              { throw SQLTableException(CantReadData);}
};

// CurrentTableRec - представление текущей записи
class CurrentTableRec : public TableRec {
public:
    CurrentTableRec(THandle tab) : TableRec(tab);
    void SetFieldValue(const char * FieldName, SQLValue * pValue)
    SQLValue * GetFieldValue(const char * FieldName)
};

// SQLTable - представление таблицы
class SQLTable {
    THandle tab;
public:
    // Конструктор открывает таблицу из файла с заданным именем
    explicit   SQLTable(const char * FileName)

    // Деструктор закрывает таблицу
              ~SQLTable()

    // MoveFirst - устанавливает указатель файла на первую запись
    //(если она есть) и считывает запись в буфер текущей записи.
    //Если таблица пуста, то состояние буфера текущей записи не
    //определено. При этом функции AfterLast и BeforeFirst выдают
    //значение true.
    bool   MoveFirst()

    // MoveNext - устанавливает указатель файла на следующую в
    //файле запись (если она есть) и считывает запись в буфер
    //текущей записи.. Если буфер уже находился на последней
    //записи, то он переходит в состояние "после последней", в
    //котором содержимое буфера не определено. При этом функция
    //AfterLast выдает значение true.
    bool   MoveNext()

    // BeforeFirst - выдает значение true, если таблица пуста,
    //иначе выдается значение false.
    bool   BeforeFirst()

    // AfterLast - Функция выдает значение true, если таблица пуста
    //или если в состоянии "на последней записи" выполняется
```

```
    //операция MoveNext, иначе выдается значение false.
    bool   AfterLast()

    // GetCurrentRec - возвращает буфер текущей записи таблицы
    TableRec * GetCurrentRec()

    // GetNewRec - очищает и возвращает буфер текущей записи
    //таблицы
    TableRec * GetNewRec()

    // UpdateCurrent - записывает содержимое буфера текущей записи
    //в файл
    void   UpdateCurrent()

    // Append - добавляет новую запись в таблицу
    void   Append()
};

// FieldDesc - описание атрибутов отношения (таблицы):
//имя атрибута, тип, длина
struct FieldDesc {
        std::string m_Name;
        EFieldType  m_Type;
        int         m_Length;
        FieldDesc(const char * Name, EFieldType Type, int len) :
                  m_Name(Name), m_Type(Type), m_Length(len) {}
};

// TableDef - описание отношения (структура таблицы)
class TableDef : public std::vector<FieldDesc> {};

// Функция создает новую таблицу с заданным именем и структурой.
// При этом создается новый файл в текущей директории, в заголовке
//которого сохраняется структура таблицы
void CreateTable(const char * TableName, TableDef& ts);

// Удаляет таблицу с заданным именем
void DeleteTable(const char * TableName);

}; // end of namespace ModelSQL

#endif
```

Все операции с таблицами (кроме создания и удаления таблицы) производятся с помощью методов класса SQLTable, находящегося в пространстве имён ModelSQL. Таким образом, гарантируется, что пользователь-программист не может работать с таблицами иначе, чем через явно описанный интерфейс. Для создания новой таблицы используется функция CreateTable, которая в качестве параметра получает имя таблицы и описание её структуры. Ниже приводится пример работы с таблицами БД.

```
#include "MSQLTable.h"

using namespace ModelSQL;

TableDef ts;
```

```
// Заполняем структуру таблицы
ts->push_back(FieldDesc("Name", EFTText, 10));
ts->push_back(FieldDesc("Age", EFTLong));

// Создаём новую таблицу
ModelSQL::CreateTable("mytable", ts);

// Объявление необходимых переменных
std::string text;
long number;
TableRec* Rec;
SQLValue* Field1;
SQLValue* Field2;

// Открываем созданную таблицу и добавляем новую запись
SQLTable Table("mytable");

// Создаём текстовое и целое поле записи
Field1 = new SQLTextValue("Sasha");
Field2 = new SQLLongValue(20);

// Создаём новую запись
Rec = Table.GetNewRec();

// Добавляем созданные поля
Rec->SetFieldValue("Name", Field1);
Rec->SetFieldValue("Age", Field2);

// Добавляем в таблицу созданную запись
Table.Append;

// Продолжаем работать с таблицей...

// Удаляем временные объекты
delete Field2;
delete Field1;
delete Rec;
```

Следующий фрагмент считывает и модифицирует первую запись таблицы.

```
#include "MSQLTable.h"

using namespace ModelSQL;

// ...

// Перемещаемся на первую запись
Table.MoveFirst();

// Считываем запись
Rec = Table.GetCurrentRec();

// Выбираем значения текстового и целого поля
Field1 = Rec->GetFieldValue("Name");

Field1->GetValue(text);
Field2 = Rec->GetFieldValue("Age")->GetValue(number);
```

```
Field2->GetValue(number);
delete Field2;
delete Field1;

// Что-то делаем, например, модифицируем целое поле
if (text == "Sasha") number += 1;

Field1 = new SQLTextValue(text);
Field2 = new SQLLongValue(number);

// Обновляем поля в записи
Rec->SetFieldValue("Name", Field1);
Rec->SetFieldValue("Age", Field2);

// Обновляем текущую запись таблицы
Table.UpdateCurrent();

// ...

delete Field2;
delete Field1;
delete Rec;
```

3. Описание модельного языка SQL

Язык SQL — простой и достаточно мощный язык взаимодействия пользователя с реляционными базами данных, разработанный фирмой IBM. Все необходимые пользователю операции, совершаемые над реляционными базами данных, могут задаваться с помощью SQL-предложений.

Модельный язык SQL будет включать лишь шесть основных предложений стандарта языка SQL, возможности которых также будут существенно ограничены. Это:

SELECT — непосредственно запрос к базе данных, обеспечивает выборку данных;

INSERT — вставить новую строку данных в таблицу;

UPDATE — обновить значения данных в существующей строке;

DELETE — удалить строку из таблицы;

CREATE — создать таблицу;

DROP — уничтожить таблицу.

Каждое предложение модельного SQL будет относиться только к одной таблице, имя которой в нем указано. Получив предложение, интерпретатор должен открыть указанную таблицу, выполнить предложение, запомнить результат и закрыть таблицу. Результатом выполнения каждого предложения (кроме DROP) будет вообще говоря новая таблица, которая в некотором внутреннем представлении должна быть передана клиенту в качестве ответа на запрос. Если же по каким-либо причинам ответить на запрос не удалось, клиент должен получить сообщение о причине неудачи.

<SQL-предложение> ::= <SELECT-предложение> |
<INSERT-предложение> | <UPDATE-предложение> |

<DELETE-предложение> | <CREATE-предложение> |
<DROP-предложение>

<SELECT-предложение> ::=
SELECT <список полей> FROM <имя таблицы> <WHERE-клауза>

<список полей> ::= <имя поля> { , <имя поля> } | *
<имя таблицы> ::= <имя>

<имя поля> ::= <имя>

<имя>::= <идентификатор языка Си>

Получив SELECT-предложение, интерпретатор выбирает из таблицы <имя таблицы> перечисленные поля в тех строках, которые удовлетворяют WHERE-клаузе.

* — обозначает все поля таблицы.

Результатом выполнения SELECT-предложения является новая, в общем случае меньшая по размерам (возможно пустая) таблица, состоящая из отобранных строк и столбцов (полей).

<INSERT-предложение> ::= INSERT INTO <имя таблицы>
(<значение поля> { , <значение поля> })

<значение поля> ::= <строка> | <длинное целое>

<строка> ::= '<символ> {<символ>}'

<символ> ::=
<любой, представляемый в компьютере символ, кроме апострофа '>

В соответствии с INSERT-предложением, интерпретатор вставляет в конец таблицы <имя таблицы> строку с перечисленным значением полей. Количество полей и их значения должны соответствовать структуре таблицы. В противном случае выдается сообщение об ошибке.

Результатом выполнения INSERT-предложения является новая, увеличенная на одну строку таблица.

<UPDATE-предложение> ::= UPDATE <имя таблицы> SET
<имя поля> = <выражение> <WHERE-клауза>

В ответ на UPDATE-предложение интерпретатор заменяет в каждой строке таблицы <имя таблицы>, удовлетворяющей WHERE-клаузе, значение поля <имя поля> на значение заданного в предложении выражения. Тип выражения должен совпадать с типом поля, в противном случае выдается сообщение об ошибке.

Результатом выполнения UPDATE-предложения является таблица с измененными значениями указанных полей.

<DELETE-предложение> ::=
DELETE FROM <имя таблицы> <WHERE-клауза>

При выполнении DELETE-предложения интерпретатор удаляет из таблицы <имя таблицы> все строки, удовлетворяющие WHERE-клаузе.

Результат выполнения DELETE-предложения — вообще говоря урезанная исходная таблица.

<CREATE-предложение> ::=
CREATE TABLE <имя таблицы> (<список описаний полей>)

<список описаний полей> ::=
<описание поля> { , <описание поля> }

<описание поля> ::= <имя поля> <тип поля>

<тип поля> ::= TEXT (<целое без знака>) | LONG

В результате выполнения CREATE-предложения создается новая пустая таблица <имя таблицы>, структура которой определяется списком описаний полей. Этот список по сути является заголовком таблицы.

В таблице могут храниться либо целые числа (тип LONG), либо строки определенной длины (тип TEXT). Допустимая длина строки задается целым числом в круглых скобках.

<DROP-предложение> ::= DROP TABLE <имя таблицы>

Выполнение DROP-предложения сводится к удалению таблицы <имя таблицы> из базы данных (и соответствующего файла из файловой системы).

<WHERE-клауза> ::=
WHERE <имя поля типа TEXT> [NOT] LIKE <строка-образец> |
WHERE <выражение> [NOT] IN (<список констант>) |
WHERE <логическое выражение> |
WHERE ALL

<строка-образец> ::= <строка>

<выражение> ::= <Long-выражение> | <Text-выражение>

<список констант> ::= <строка> { , <строка> } |
<длинное целое> { , <длинное целое> }

<Long-выражение> ::=
<Long-слагаемое> { <+|-> <Long-слагаемое> }

<+|-> ::= + | -

<Long-слагаемое> ::=
<Long-множитель> { <*|/|%> <Long-множитель> }

<*|/|%> ::= * | / | %

<Long-множитель> ::= <Long-величина> | (<Long-выражение>)

<Long-величина> ::= <имя поля типа LONG> | <длинное целое>
<Text-выражение> ::= <имя поля типа TEXT> | <строка>

<логическое выражение> ::=
<логическое слагаемое> { OR <логическое слагаемое> }

<логическое слагаемое> ::=
<логический множитель> { AND <логический множитель> }

<логический множитель> ::= NOT <логический множитель> |
(<логическое выражение>) |
(<отношение>)

<отношение> ::= <Text-отношение> | <Long-отношение>

<Text-отношение> ::=
<Text-выражение> <операция сравнения> <Text-выражение>

<Long-отношение> ::=
<Long-выражение> <операция сравнения> <Long-выражение>

<операция сравнения> ::= = | > | < | >= | <= | !=

WHERE-клауза при выполнении предложений SELECT, UPDATE и DELETE играет роль фильтра: требуемые действия производятся не со всеми строками заданной таблицы, а только с теми из них, поля которых удовлетворяют условиям WHERE-клаузы.

Первая альтернатива WHERE-клаузы — **LIKE-альтернатива** — позволяет выбрать строки, поля которых (они должны быть текстового типа) соответствуют (не соответствуют в случае с NOT) строке-образцу. В строке-образце наряду с обычными символами выделены специальные символы. Это:

% — обозначает любую последовательность из нуля или более символов;

_ — обозначает любой одиночный символ;

[ ] — обозначает любой одиночный символ из перечисленных в квадратных скобках. Например, [abcdef]. В квадратных скобках можно задавать и диапазон допустимых символов. Например, [a-f].

[^] — обозначает любой одиночный символ, не принадлежащий перечисленным в квадратных скобках. Например, [^abcdef] или [^a-f].

Вторая альтернатива WHERE-клаузы — **IN-альтернатива** — содержит выражение текстового или целого типа. Роль переменных в этом выражении играют поля строки таблицы. IN-альтернатива позволяет выбрать те строки, для которых выражение принимает (не принимает в случае с NOT) одно из значений, перечисленных в списке констант <список констант>. Тип констант из списка должен совпадать с типом выражения, а, следовательно, и с типом имен полей, в него входящих.

Третья альтернатива WHERE-клаузы — **BOOL-альтернатива** — содержит выражение логического типа, содержащее, как и в IN-альтернативе, имена полей строки в качестве переменных. Эта альтернатива позволяет выбрать строки, для которых <логическое выражение> истинно.

Логические операции NOT, AND, OR имеют обычный смысл.

Четвертая альтернатива WHERE-клаузы — **ALL-альтернатива** — говорит о том, что фильтрацию строк проводить не нужно.

Примеры предложений модельного SQL

1. Создать таблицу Students, состоящую из четырех полей: поля First_name типа TEXT длины 10, поля Surname типа TEXT длины 15, поля Age типа LONG и поля Phone типа TEXT длины 9.

```
CREATE TABLE Students (First_name TEXT (10),
                       Surname TEXT (15),
                       Age LONG,
                       Phone TEXT (9) )
```

В результате создается новая пустая таблица Students с заголовком:

First_name	Surname	Age	Phone

Её структура фиксируется в начальных зонах файла таблицы.

2. Внести в таблицу Stulents сведения о студентах Иванове, Петрове, Федорове и Захарове.

```
INSERT INTO Students ( 'Sergey', 'Ivanov', 18, '145-45-45' )
INSERT INTO Students ( 'Alexey', 'Petrov', 20, '343-65-45' )
INSERT INTO Students ( 'Andrey', 'Fedorov', 23, '123-45-18' )
INSERT INTO Students ( 'Alexandre', 'Zaharov', 20, '345- 33-33' )
```

После выполнения этих предложений таблица будет выглядеть так:

First_name	Surname	Age	Phone
Sergey	Ivanov	18	145-45-45
Alexey	Petrov	s20	343-65-45
Andrey	Fedorov	23	123-45-18
Alexandre	Zaharov	20	450- 33-33

3. Найти имена и фамилии студентов в возрасте от 18 до 29 лет.

```
SELECT First_name, Surname FROM Students WHERE Age IN (18, 19, 20)
```

Результатом выполнения предложения должна быть таблица:

First_name	Surname
Sergey	Ivanov
Alexey	Petrov
Alexandre	Zaharov

4. Выбрать всю информацию о студентах, телефоны которых оканчиваются на 45.

SELECT * FROM Students WHERE Phone LIKE '%-%-45'

Результат выполнения предложения будет такой:

First_name	Surname	Age	Phone
Sergey	Ivanov	18	145-45-45
Alexey	Petrov	20	343-65-45

5. Выбрать всю информацию о студентах, телефоны которых содержат 45.

SELECT * FROM Students WHERE Phone LIKE '%45%'

В результате будет получена следующая таблица:

First_name	Surname	Age	Phone
Sergey	Ivanov	18	145-45-45
Alexey	Petrov	20	343-65-45
Andrey	Fedorov	23	123-45-18
Alexandre	Zaharov	ы20	450- 33-33

6. Найти телефон студента Иванова.

SELECT Phone FROM Students WHERE Surname = 'Ivanov'

Результат:

Phone
145-45-45

7. Найти фамилии всех студентов.

SELECT Surname FROM Students WHERE ALL

Результат получится таким:

Surname
Ivanov
Petrov
Fedorov
Zaharov

8. Найти информацию о студентах, имена которых начинаются на любую из первых трех букв латинского алфавита, вторая буква их имен не совпадает с m, n, o, третья буква — любая, а четвертая — x.

SELECT * FROM Students WHERE First_name LIKE '[ABC][^mno]_x%'

Ответ на такой запрос будет следующим:

First_name	Surname		Age	Phone
Alexey	Petrov	20	343-65-45	
Alexandre	Zaharov		20	450- 33-33

9. Найти фамилии и телефоны всех студентов старше 19 лет, фамилии которых начинаются с букв второй половины латинского алфавита.

SELECT Surname, Phone FROM Students WHERE (Age > 19) AND (Surname > 'M')

В результате получится следующая таблица:

Surname	Phone
Petrov	343-65-45
Zaharov	450-33-33

10. Увеличить значение поля «возраст» у всех студентов на 1.

```
UPDATE Students
    SET Age = Age + 1
    WHERE ALL
```

Результат выполнения предложения будет таким:

First_name	Surname	Age	Phone
Sergey	Ivanov	19	145-45-45
Alexey	Petrov	21	343-65-45
Andrey	Fedorov	24	123-45-18
Alexandre	Zaharov	21	450- 33-33

11. Удалить таблицу Students из базы данных.

```
DROP TABLE Students
```

В результате таблица Students будет удалена из базы данных, а соответствующий ей файл — из файловой системы.

ПРИЛОЖЕНИЕ

Пример реализации объектно-ориентированногоинтерфейса для работы с сокетами

Файл "sock_wrap.h"

```
#ifndef __SOCK_WRAP_H__
#define __SOCK_WRAP_H__

#include <string>
#include <iostream>
#include <io.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

namespace ModelSQL {

// Exception - класс исключений, генерируемых библиотекой
class Exception {
protected:
    int     m_ErrCode;
public:
                Exception(int errcode) : m_ErrCode(errcode) {}
    void        Report();
    virtual std::string GetMessage() = 0;
};

// SocketException - класс исключений
class SocketException : public Exception {
    static char * m_Message[];
public:
    enum SocketExceptionCode {
         ESE_SUCCESS,
         ESE_SOCKCREATE,
         ESE_SOCKCONN,
         ESE_SOCKILLEGAL,
         ESE_SOCKHOSTNAME,
         ESE_SOCKSEND,
         ESE_SOCKRECV,
         ESE_SOCKBIND,
         ESE_SOCKLISTEN,
         ESE_SOCKACCEPT,
    };
    SocketException(SocketExceptionCode errcode) :
Exception(errcode) {}
    std::string  GetMessage();
};

// SocketAddress - базовый абстрактный класс для представления
//сетевых адресов
class SocketAddress {
protected:
```

```
    struct sockaddr * m_pAddr;
public:
                SocketAddress () : m_pAddr(NULL) {}
    virtual     ~SocketAddress () {}
    virtual int GetLength() = 0;
    virtual     SocketAddress * Clone() = 0;
                operator struct sockaddr * ();
};

// UnSocketAddress - представление адреса семейства AF_UNIX
class UnSocketAddress : public SocketAddress {
public:
        UnSocketAddress (const char * SockName);
    int GetLength ();
        SocketAddress * Clone();
        ~UnSocketAddress ();
};

// InSocketAddress - представление адреса семейства AF_INET
class InSocketAddress : public SocketAddress {
public:
        InSocketAddress (const char * HostName, short PortNum)
    int GetLength ()
        SocketAddress * Clone()
        ~InSocketAddress ()
};

// BaseSocket - базовый класс для сокетов
class BaseSocket {
public:
 explicit BaseSocket(int sd = -1, SocketAddress * pAddr = NULL):
                        m_Socket(sd), m_pAddr(pAddr) {}
    virtual         ~BaseSocket();
    void            Write(void * buf, int len);
    void            PutChar(int c);
    void            PutString(const char * str);
    void            PutString(const std::string& s)
    int             Read (void * buf, int len);
    int             GetChar();
    std::string     GetString();
    int             GetSockDescriptor();
protected:
    int             m_Socket;
    SocketAddress * m_pAddr;
    void            CheckSocket()
};

// ClientSocket - базовый класс для клиентских сокетов
class ClientSocket: public BaseSocket {
public:
    void            Connect();
};

// ServerSocket - базовый класс для серверных сокетов
class ServerSocket: public BaseSocket {
public:
    BaseSocket *    Accept();
protected:
```

```
    void            Bind();
    void            Listen(int BackLog);
    virtual void    OnAccept (BaseSocket * pConn) {}
};

// UnClientSocket - представление клиентского сокета семейства
//AF_UNIX
class UnClientSocket: public ClientSocket {
public:
    UnClientSocket(const char * Address)
};

// InClientSocket - представление клиентского сокета семейства
//AF_INET
class InClientSocket: public ClientSocket {
public:
     InClientSocket(const char * HostName, short PortNum);
};

// UnServerSocket - представление серверного сокета семейства
//AF_UNIX
class UnServerSocket: public ServerSocket {
public:
    UnServerSocket(const char * Address);
};

// InServerSocket - представление серверного сокета семейства
//AF_INET
class InServerSocket: public ServerSocket {
public:
   InServerSocket(short PortNum) throw (SocketException);
};

}; // конец namespace ModelSQL

#endif
```

ЛИТЕРАТУРА

[1] К. Дейт. Введение в системы баз данных. — М.: Вильямс, 2001.

[2] М. Грабер. Введение в SQL. — М.: Изд-во ЛОРИ, 1996.

[3] С. Дунаев. UNIX. System V. Release 4.2. Общее руководство. — М.: «Диалог-МИФИ», 1996.

[4] И.А. Волкова, Т.В. Руденко. Формальные грамматики и языки. Элементы теории трансляции. — М.: Изд-во МГУ, 1996.

СОДЕРЖАНИЕ