Practice 6.

WORKING WITH DATABASES IN Qt.

The purpose of the work is to learn how to create databases and work with them in Qt.

1.Theoretical introduction

A database is a system for storing records (corteges) organized in the form of tables (relations). The database can contain from one to several hundred tables that are related to each other. A table consists of a set of rows (corteges) and columns (attributes). The columns of the table are named, and each column (attribute) is assigned a type and / or range (subset). The rows in a database table are called records, and the cells that the record is divided into are called fields.

A *primary key* is a unique identifier for a record, which can represent not only one column, but also a whole combination of columns (composite key).

The user can perform many different operations with tables: *add, modify and delete records, search,* etc. To compose differen queries, the SQL language (StructuredQueryLanguage - the language of structured queries) is used, which makes it possible not only to make queries and change data, but also to create new databases.

The Qt library contains drivers for working with the following
DBMS: QDB2 –IBM DB2 version 7.1 or later;
QIBASE –Borland InterBase;
QMYSQL –MySQL;
QOCI –Oracle;
QODBC –ODBC (including Microsoft SQL Server);
QPSQL –PostgreSQL;
QSQLITE – SQLite version 3 or higher;
QSQLITE2 –SQLite version2;
QTDS –Sybase Adaptive Server.

It is also possible to write your own driver for working with the database, if the existing ones do not suit you.

The Qt Open Source Edition lacks support for the commercial Oracle, Sybase and DB2 databases. Drivers for them are distributed under a license that is not compatible with the GPL. SQLite is a compact embedded relational database management system. The library's source code has been released into the public domain. SQLite does not have a client-server architecture. That is, the database is not a separately working process with which the program interacts. SQLite is the library with which your program links and thus becomes part of the program. That is, it is a DBMS that does not require a DB server and the client itself.

2.Preparation for connecting the OBD

1) Run QtCreator (... \ Qt \ Tools \ QtCreator \ bin \ qtcreator.exe).

Use the wizard.

Select File → New File or Project ..., and there Application → QtWidget Project (Fig.1).

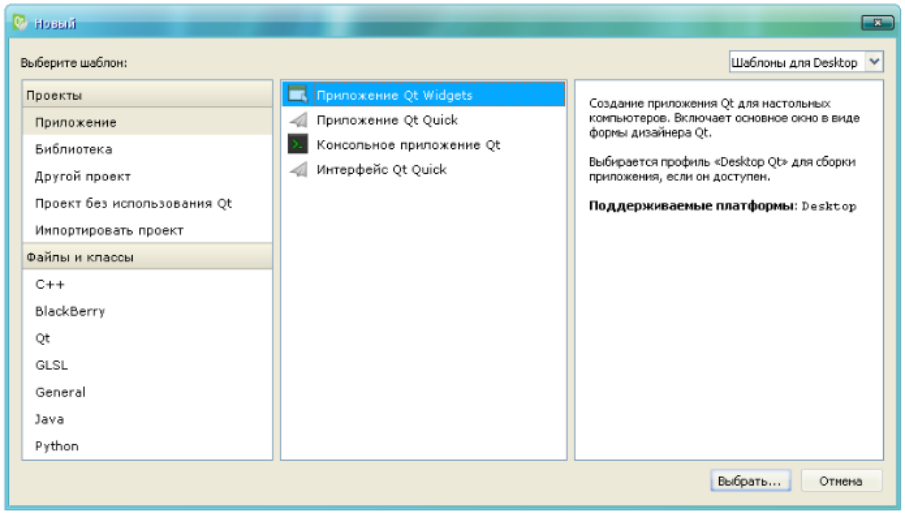

Fig.1.

2) Give it some meaningful name, indicate the location of the project, click ***Next*** (three times) and *Finish*.

3) It is convenient to navigate the project using the navigation panel (Fig.2):

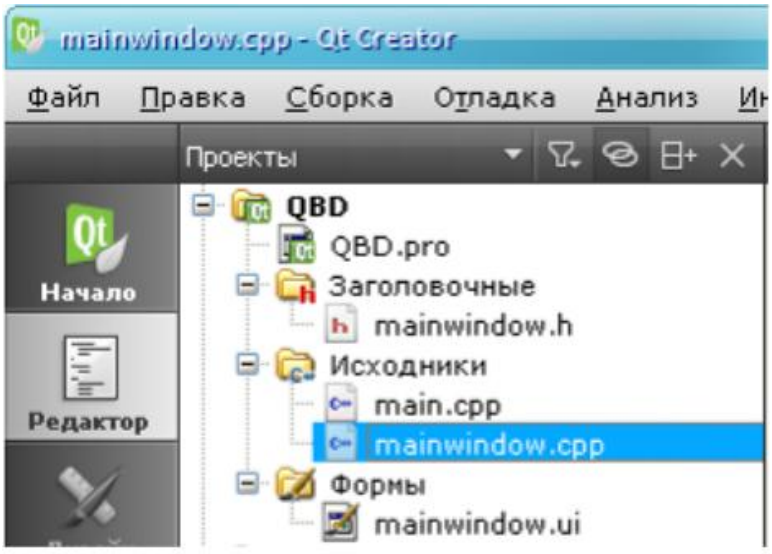

Fig.2

4) Include SQL.

For using databases, Qt provides a separate Qt Sql module. To connect it, you need to add in the project file (with the .pro extension):

```
QT+=sqlwidgets
```

and also, in order to work with the classes of this module, you need to include them in the header file of the main window (mainwindow.h):

```
#include<QtSql>
#include<QSqlDatabase>
```

```
#include<QSqlError>
#include<QSqlQuery>
#include<QMessageBox>//to display trace messages
```

Run and test the application

2. Connecting to the database and executing SQL queries

1) Establishing a connection to the database.

Let's add variables and a function to the class of the main window (file mainwindow.h, section private):

```
//database
QSqlDatabasedb;
//data model
QSqlTableModel*model;
// function for connecting to the database
boolcreateConnection();
```

Let's create a function in the mainwindow.cpp file to implement a connection to the database.

```
boolMainWindow::createConnection()

{return true;}
```

Let's connect it in the constructor (after `ui->setupUi(this);`):

```
createConnection();
```

Write all further code inside the createConnection () function. To connect to the database, you should specify the name of the SQL driver, for example:

```
QSqlDatabasedb=QSqlDatabase::addDatabase("QSQLITE");
```

Then you can specify the server name, database name, username and password:

```
//db.setHostName("localhost");
//or, for example, "my1.server.ru"
//db.setDatabaseName("mydb1");
//db.setUserName("root");
//db.setPassword("mypassword");
```

We can omit all these parameters. Instead, let's write:

```
db.setDatabaseName(":memory:");
```

Note: The predefined name ": memory:" allows you to place a temporary database in RAM.

After all the connection parameters are set, you can open the connection (open () method).

If the connection could not be established, then it would be nice to find out the error description and inform the user about it:

```
if(!db.open())
{QMessageBox::critical(0,qApp->tr("Cannotopendatabase"),
qApp->tr("Unable to establish a database connection.\n"
"This example needs SQLite support. Please read"
"the Qt SQL driver documentation for information how"
"to build it.\n\n"
"Click Cancel to exit."),
QMessageBox::Cancel);
returnfalse;}
```

2) Primary filling of the database.

You can use the QSqlQuery class to execute SQL commands after a connection is established.

Queries (commands) are formatted as a regular string, which is passed to the method

`QSqiQuery::exec()`

If the connection is established, then you can execute any SQL query, for example, create a table:

```
if(db.tables().empty())//checking the presence of tables
{//QMessageBox::critical(0,tr("Error"),tr("Databaseisempty"));
```

//table creation:

```
QSqlQueryquery;query.exec("CREATE TABLE person(id integer PRIMARY KEY
NOT NULL,"
```

This is important: the QSqlQuery class can be used to execute Data Manipulation Language (DML) expressions such as SELECT, INSERT, UPDATE and DELETE, and DDL (Data Definition Language) expressions such as CREATE TABLE.

```
"firstnamevarchar(20),lastnamevarchar(20))");
// filling the table:
query.exec("INSERT INTO person VALUES(1,'Alina','Bichenko')");
query.exec("INSERT INTO person VALUES(2,'Kate','Mikova')");
query.exec("INSERT INTO person VALUES(3,'Olga','Skalskaya')");
query.exec("INSERT INTO person VALUES(4,'Alexander','Panchenko')");
query.exec("INSERT INTO person VALUES(5,'Victorya','Silivina')");
query.exec("INSERT INTO person VALUES(6,'July','Tokmakova')");}
```

Data can be added in a different way:

```
query.prepare("INSERT INTO person(id,firstname,lastname,ball)"
"VALUES(?,?,?,?)");
query.addBindValue(7);
query.addBindValue("Bart");
query.addBindValue("Simpson");
query.addBindValue(1);
query.exec();
```

You can simply use the substitution arguments that QString provides:

```
query.exec(QString("INSERT INTO person(id,firstname,lastname,ball)
VALUES(%1,'%2','%3',%4);").arg("8").arg("Bilbo").arg("Bagins").arg("5"));
```

4) Displaying the contents of the table on the screen.

Switch to visual editing mode (by double clicking on mainwindow.ui).

Drag a *TableView* element onto the form (Fig.3):

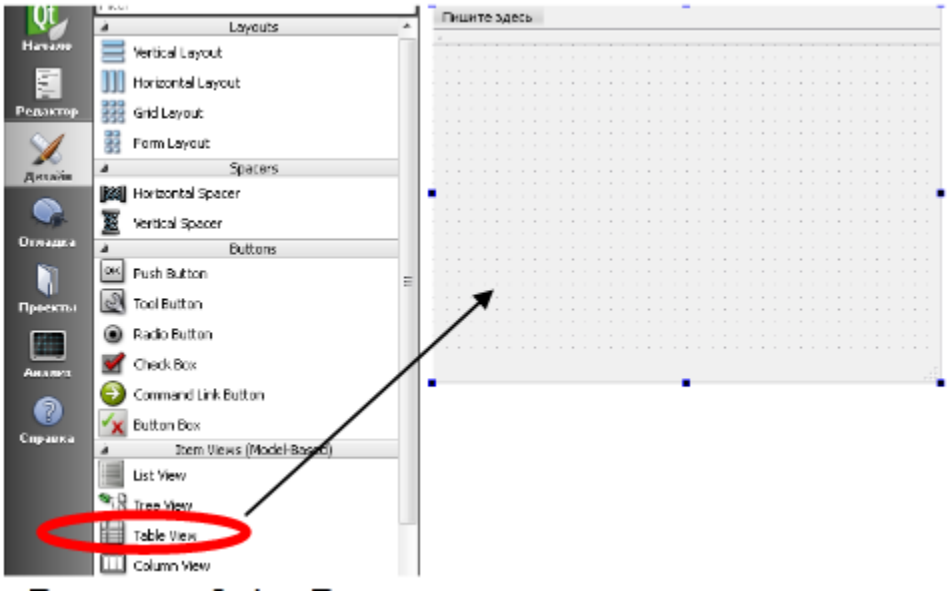

Fig.3

Go back to mainwindow.cpp. Let's link the data model to the table on the form: // declare the data model

```
model=newQSqlTableModel(this);
// specify a table from the database
model->setTable("person");
// enter data into the model // if successful
if(model->select()){
// transfer data from the model to
tableViewui->tableView->setModel(model);
// set the line height to the text
ui->tableView->resizeRowsToContents();
```

```
// first column header
model->setHeaderData(0,Qt::Horizontal,tr("ID"));
// second column header
model->setHeaderData(1,Qt::Horizontal,tr("Имя"));
// passing control to an element
tableViewui->tableView->setFocus();}
```

Run and test the application.

5) Layout and display.

Using (in design mode) layout  ,
achieve a fixed arrangement of elements on the screen.
Go to the properties of the *tableView* element and find *horizontalHeaderStretchLastSection* and put **"v"** - it will automatically stretch the columns to the full length of the *tableView*.
Change the display of the names of all columns, for example:

```
model->setHeaderData(2,Qt::Horizontal,QObject::tr("Surname"));
```

If you go to the properties of the *tableView* element and activate the *SortingEnable*, this will allow you to sort the column by clicking on its header.
Run and test the application.

6) Additional information.
The *QSqlTableModel class* provides the following editing strategies
(*set with setEditStrategy* ()):
OnRowChange — writes data as soon as the user moves to another row in the table.
OnFieldChange — writes after the user navigates to another cell in the table.
OnManualSubmit – writes data when the *submitAll* () slot is called. If the *revertAll* () slot is called, then the data is returned to its original state. In the program code, it looks like this:

```
model->setEditStrategy(QSqlTableModel::OnFieldChange);
```

If an error occurs while executing the request, then the lastError () method allows you to display its description on the screen:

```
if(!query.isActive())
QMessageBox::warning(this,tr("DatabaseError"),
query.lastError().text());
```