

Using ODBC with Red Hat Database

by Permaine Cheung (pcheung@redhat.com) of Red Hat, Inc.

ODBC (Open Database Connectivity), a standardized API that provides a product-neutral interface between frontend applications and database servers, enables applications to be transportable between database servers from different vendors, as long as the database schema is the same. This document provides an overview of how to use ODBC with Red Hat Database, a powerful and robust open-source database solution.

Introduction

ODBC (Open Database Connectivity) is a standardized API that can be used to access data stored in various database management systems. It provides a product-neutral interface between frontend applications and database servers, enabling a developer to write applications that are transportable between database servers from different vendors.

ODBC inserts a middle layer, called a database driver, between an application and the database management system. The purpose of this layer is to enable access to the database backends and to translate the data queries and commands in an application into commands that the database management system understands. In order to achieve these goals, both the application and the database management system must be capable of issuing ODBC commands and the database management system must be capable of responding to them. Once an ODBC application is written, it should be able to connect to any backend database by using the appropriate ODBC driver, regardless of the vendor, as long as the database schema is the same.

This document provides an overview of how to use ODBC with Red Hat Database, a powerful and robust open-source database solution. We explain how to obtain, install, and configure the ODBC driver, then we use a sample application to illustrate some basic aspects of ODBC.

Getting Ready to Use ODBC with Red Hat Database

Getting the ODBC Driver

As Red Hat Database (RHDB) is powered by PostgreSQL, Red Hat Database ships with `psqlODBC`, the PostgreSQL ODBC 2.5 driver. The `psqlODBC` driver has been modified from the source code for PostODBC, the ODBC driver developed for Postgres95, and is distributed under the terms and conditions of the GNU Library General Public License. There are other ODBC drivers available for Linux for which you can get the source code and build the drivers yourself. For more information on downloading and building ODBC from source, see PostgreSQL's ODBC home page at <http://odbc.postgresql.org>.

Installing the ODBC Driver

The `psqlODBC` ODBC driver that is included with Red Hat Database is in the `rh-postgresql-odbc` package. It provides the driver and an ODBC catalog-extensions file for Red Hat Database. If you would like to develop applications that access RHDB databases, you also need to install the `rh-postgresql-devel` package, which provides the include files and library necessary for writing applications using ODBC functions. Refer to the *Red Hat Database Getting Started/Installation Guide* for information about the packages.

After installing the package(s), you need to add the ODBC catalog extensions to the database template. (Refer to the *Red Hat Database Administrator and User's Guide* for information about templates.) The file `/usr/share/pgsql/odbc.sql` (in the default installation layout) contains the appropriate definitions. The catalog extensions are functions required by the ODBC standard, but are not supplied by PostgreSQL by default. You can add the extensions to the database template as follows:

```
$ psql -d template1 -f /usr/share/pgsql/odbc.sql
```

Specifying `template1` as the target database ensures that all subsequent new databases will contain these name definitions.

Configuration Files

The `.odbc.ini` file contains the access information for the `psqlODBC` driver, and it must be placed in your home directory. The `.odbc.ini` file has two main sections:

- The first section is called `[ODBC Data Sources]`. This is a list of arbitrary names and descriptions for each database you want to access.
- The second section contains the data source specification. There will be one of these specifications for each name given in the first section. Every specification must be labeled with the name given in `[ODBC Data Sources]` and must contain at least the following entries:

```
Database=DatabaseName
Servername=host
```

Here is a sample `.odbc.ini` file:

```
[ODBC Data Sources]
RO = Read-only Database
debug = Debugging Database
RHDB = Red Hat Database

[RO]
Database = rhdb1
Servername = localhost
ReadOnly = 1

[debug]
Database = rhdb1
Servername = machine1
Debug = 1
CommLog = 1
ReadOnly = 0
Username = pcheung
Password = "pcheung1"
Port = 5432
Driver = /usr/lib/libpsqlodbc.so

[RHDB]
Servername = localhost
Database = basketball
ReadOnly = No
CommLog = 1
Debug = 1
```

In this sample `.odbc.ini` file, the first section specifies three data source names (DSN): `RO`, `debug`, and `RHDB`.

For the `[RO]` DSN, writing to the `rhdb1` database is prohibited by `ReadOnly` being set to `on (1)`. The `userid` and `password` for this DSN will have to be handled within the application itself.

The `[debug]` DSN specifies a connection to the `rhdb1` database as user `pcheung` and password `pcheung1`, connecting to the server running on `machine1` listening to port `5432`, and using the driver found in `/usr/lib/libpsqlodbc.so`. Port `5432` is the default port number the server listens on. Modifications can be made to the database because `ReadOnly` is set to `off (0)`. Debug information and communications to and from the backend are logged. When `Debug` is `on`, debug information goes to the `/tmp/mylog_<userid><pid>.log` file. Similarly, when `CommLog` is turned `on`, the communications to and from the backend are logged to the file `/tmp/psqlodbc_<userid><pid>.log`. The `CommLog` contains information about the connection, the DSN, the queries being carried out, and so on. Both `Debug` and `CommLog` are `off` by default.

If you want to use another ODBC driver, modify the `Driver` setting for that DSN.

The `[RHDB]` DSN connects to the database `basketball` with the database server running local, and with debug and communications output generated. You can also turn a setting `on` or `off` by giving it a value of `yes` or `no`. For example, the `[RHDB]` DSN setting `ReadOnly = No` enables a user to update the `basketball` database.

Setting Up Red Hat Database / PostgreSQL

Permission to access the databases from a client machine is required in order for ODBC to connect to it. To enable access to the database, modify the `pg_hba.conf` file appropriately. TCP/IP connections are required in order for ODBC to connect to the databases; you can do this by making appropriate changes to the `tcpip_socket` setting in the `postgresql.conf` file. Restart `postmaster` after all modifications are complete to invoke the new settings. For more information on granting database access permission, enabling TCP/IP connections and restarting the `postmaster`, see the *Red Hat Database Administrator and User's Guide*.

Sample Application

In this section we examine various aspects of ODBC using a sample program. The topics that will be covered are:

- Loading the driver and connecting to a database
- Performing queries and updates
- Retrieving results
- Transactions
- Closing a connection.

Database Schema

For the sample code, we assume a database called `basketball`. The `basketball` database contains one table called `players`, which contains a player's name and team.

Create the table by issuing the following SQL statements:

```
-- create the players table
CREATE TABLE players(
    name      varchar(25),
    team      varchar(50)
);
```

Populate the table with the following data:

```
-- insert 3 records into the players table
INSERT into players VALUES ('Michael Jordan', 'Washington Wizards');
INSERT into players VALUES ('Tim Duncan', 'San Antonio Spurs');
INSERT into players VALUES ('Vince Carter', 'Toronto Raptors');
```

Loading the Driver and Connecting to a Database

When using the ODBC driver, you need to allocate an environment handle, which is an environment storage area. The allocation of the environment handle must be the first routine called by the application using the ODBC interface. After the environment handle is allocated, you use the handle to allocate a connection handle.

The declarations for the environment handle and connection handle are as follows:

```
HENV henv;    /* environment handle */
HDBC hdbc;    /* connection handle */
```

To allocate the environment handle:

```
SQLAllocEnv(&henv);
```

To allocate the connection handle associated with the environment:

```
SQLAllocConnect(henv, &hdbc);
```

When the `SQLAllocEnv` or `SQLAllocConnect` are called, the driver manager allocates a structure for storing the information about the environment and the connection.

To connect to a database, you need to specify the DSN, userid, and password. The DSN definition must exist in the `.odbc.ini` file:

```
SQLCHAR DSN[10] = "RHDB";
SQLCHAR DSN_userid[9] = "postgres";
SQLCHAR DSN_password[9] = "postgres";
SQLConnect(hdbc, DSN, SQL_NTS, DSN_userid, SQL_NTS,
           DSN_password, SQL_NTS);
```

When the `SQLConnect` statement is called, the driver manager looks for the driver to be used and checks to see if a driver has been loaded for the particular connection. If no driver is loaded in the environment for the connection, the driver manager loads the driver and allocates the environment handle (which is `hdbc` in this example).

Refer to `Initialize.h` in the *Code Samples* section for the complete example.

Performing Queries and Updates

To perform queries, you need a statement handle, which must be allocated before it is used. This is the declaration and allocation for a statement handle:

```
HSTMT hstmt;
SQLAllocStmt(hdbc, &hstmt);
```

After the statement handle is allocated, you need to prepare the statement handle so that the SQL statement is sent to the data source and compiled for execution. In this example the actual SQL query is stored in `selectStmt`, which is a `SQLCHAR` array:

```
SQLCHAR SelectStmt[255];
strcpy ((char *) SelectStmt, "SELECT * FROM players");
SQLPrepare(hstmt, SelectStmt, SQL_NTS);
```

After the SQL statement is successfully prepared, the `SQLExecute` routine executes the SQL statement:

```
SQLExecute(hstmt);
```

Changing the content of `SelectStmt` also enables you to perform updates and deletes to the table. For example, you can delete a record from the table as follows:

```
strcpy ((char *) SelectStmt, "DELETE FROM players WHERE ");
strcat ((char *) SelectStmt, "name=\'Michael Jordan\'");
SQLExecDirect(hstmt, SelectStmt, SQL_NTS );
```

The effect of `SQLExecDirect` is equivalent to issuing `SQLPrepare` and `SQLExecute`.

Retrieving Results

To traverse a query result set, you need to bind all the columns returned with variables in the application. The following will bind column 1 to the variable `cname1`, and column 2 to the variable `cname2`:

```
SQLBindCol(hstmt, 1, SQL_C_CHAR, cname1, sizeof (cname1), NULL);
SQLBindCol(hstmt, 2, SQL_C_CHAR, cname2, sizeof (cname2), NULL);
```

After you bind the columns, you can perform `SQLFetch` to retrieve the next result from the result set:

```
res = SQLFetch(hstmt);
while (res != SQL_NO_DATA_FOUND) {
    printf("%s %s \n", cname1, cname2);
    res = SQLFetch(hstmt);
}
```

Transactions

In the code that you have seen so far, each SQL statement has been atomic. That is, each statement can stand on its own and if one statement fails, others are not affected and the database is left in a consistent state. By default, ODBC is in "auto-commit" mode, where each statement is either committed as soon as it succeeds or rolled back if it fails.

There are times when multiple statements need to be grouped together into one atomic

action or transaction. Use the `SQLSetConnectOption` function to turn auto-commit mode on or off. The first statement executed after turning off auto-commit mode or after committing or rolling back a transaction starts a new transaction. The `SQLTransact` function commits or rolls back a transaction.

To turn autocommit off:

```
SQLSetConnectOption(hdbc, SQL_AUTOCOMMIT, SQL_AUTOCOMMIT_OFF);
```

After autocommit is turned off, the transaction begins with the next executable statement. To commit the transaction:

```
SQLTransact(henv, hdbc, SQL_COMMIT);
```

Replace `SQL_COMMIT` by `SQL_ROLLBACK` to terminate the current transaction and remove the changes made within that transaction.

Closing a Connection

Closing a connection is fairly simple: all you have to do is close and free the explicitly allocated handles. To disconnect from the database and free the different handles:

```
SQLFreeStmt(hstmt, SQL_CLOSE); /* free the statement handle */
SQLDisconnect(hdbc);          /* disconnect from the database */
SQLFreeConnect(hdbc);        /* free the connection handle */
SQLFreeEnv(henv);           /* free the environment handle */
```

References and Further Reading

- *Red Hat Database Getting Started / Installation Guide*
- *Red Hat Database Administrator and User's Guide*
- Red Hat Database. <http://www.redhat.com/software/database>
- PostgreSQL Global Development Group. <http://www.postgresql.org>
- PostgreSQL ODBC Driver Group. <http://odbc.postgresql.org>
- *ODBC Developers Guide*, Tom Johnston, Dec. 1995
- *The ODBC Solution: Open Database Connectivity in Distributed Environments*, Robert Signore, John Creamer, Michael O. Stegman, Jan. 1995.

Code Samples

This section lists the following code samples:

- Initialize.h
- PrepAndExec.c
- ExecDirect.c
- ViewTable.c
- Transact.c

Compiling and Running the Sample Application

This section contains the steps required to get the sample code working with Red Hat Database.

1. If they are not already installed, install the `rh-postgresql-odbc` and `rh-postgresql-devel` packages (both are required for this application to work). See the section “Installing the ODBC Driver” on page 2.
2. Add the ODBC catalog extensions (as user `postgres`) to `template1`:

```
$ psql -d template1 -f /usr/share/pgsql/odbc.sql
```
3. Create a database called `basketball` using your `userid`. If your `userid` is not allowed to create databases, talk to your database administrator or refer to the *Red Hat Database Administrator and User's Guide* for instructions on how to grant “create database” permission to a user.
4. Create a table called `players` within the `basketball` database, then insert a few records into the table using the `psql` commands described under “Database Schema” on page 5.
5. For a standard installation of Red Hat Database, the host access control file is located at `/var/lib/pgsql/data/pg_hba.conf`. Edit `pg_hba.conf` to add a record similar to the following:

```
host    all    127.0.0.1    255.255.255.255    trust
```

Refer to the *Red Hat Database Administrator and User's Guide* for a proper client authentication record to suit your configuration.

6. To accept TCP/IP connections, edit the configuration file to allow TCP/IP connections. For a standard installation of Red Hat Database, the configuration file is located at `/var/lib/pgsql/data/postgresql.conf`. Modify the value of `tcpip_socket` from `false` to `true` so that TCP/IP connections are allowed by default every time `postmaster` starts up. (Another way to this is by restarting `postmaster` with the `-i` option.)
7. Restart `postmaster` so that the changes will be effective. You can do that by issuing the following command as user `postgres`:

```
$ pg_ctl restart
```

8. Edit the `.odbc.ini` file and add the following:

```
[ODBC Data Sources]
RHDB = RedHatDatabase

[RHDB]
Servername = localhost
Database = basketball
ReadOnly = No
CommLog = 1
Debug = 1
```

9. Create files named `Initialize.h`, `PrepAndExec.c`, `ExecDirect.c`, `ViewTable.c`, and `Transact.c`. (See the *Code Sample* section.)

Remember to update the DSN definition in `Initialize.h` for your setting.

10. Compile the sample code:

```
gcc -lpsqlodbc -I/usr/include/pgsql PrepAndExec.c -o PrepAndExec
gcc -lpsqlodbc -I/usr/include/pgsql ExecDirect.c -o ExecDirect
gcc -lpsqlodbc -I/usr/include/pgsql ViewTable.c -o ViewTable
gcc -lpsqlodbc -I/usr/include/pgsql Transact.c -o Transact
```

These commands compile the code, link with the ODBC library (`psqlodbc.so`), and use the include files under the `/usr/include/pgsql` directory.

11. Show the contents of the `players` table as follows:

```
$ ViewTable
```

You should see the following output:

```
Connection Parameters: DSN='RHDB', UID='postgres', PWD='postgres'
```

```
Handles initialized.
```

```
Content of players:
*****
Michael Jordan -- Washington Wizards
Tim Duncan -- San Antonio Spurs
Vince Carter -- Toronto Raptors
```

- 12.** Use the `PrepAndExec` command to prepare and execute the SQL statement that inserts a record into the `players` table:

```
$ PrepAndExec "Insert into players values('Kobe Bryant', \
  'Portland Trail Blazers')"
```

You can execute `ViewTable` again to see the updated content of the `players` table.

- 13.** Use the `ExecDirect` command to execute directly the SQL statement that modifies a record in the `players` table:

```
$ ExecDirect "Update players set team='LA Lakers' \
  where name='Kobe Bryant'"
```

You can execute `ViewTable` again to see the updated content of the `players` table.

- 14.** `Transact.c` is an example that contains a transaction with the following SQL statements:

```
INSERT into players VALUES ('Tracy McGrady', 'Orlando Magic')
DELETE FROM players WHERE name='Michael Jordan'
```

To execute the SQL statements in the transaction and rollback at the end of the transaction:

```
$ Transact 0
```

To check that the changes are not applied to the `players` table.

```
$ ViewTable
```

To execute the SQL statements in a transaction and commit them at the end of the transaction:

```
$ Transact 1
```

To see that the modifications are applied to the `players` table:

```
$ ViewTable
```

Initialize.h

```
#include <stdio.h>
#include <stdlib.h>
#include "iodbc/iodbc.h"
#include "iodbc/isqlext.h"

SQLCHAR DSN[10] = "RHDB";

// The following should be changed to your userid and password
// and the length of the variable should be the actual length + 1
SQLCHAR DSN_userid[9] = "postgres";
SQLCHAR DSN_password[9] = "postgres";

int Initialize(HENV * phenv, HDBC * phdbc, HSTMT * phstmt);
void cleanup(HENV henv, HDBC hdbc, HSTMT hstmt);

// Allocate the environment, connection, statement handles
int Initialize(HENV * phenv, HDBC * phdbc, HSTMT * phstmt)
{
    int res;

    // Allocate an environment handle
    res = SQLAllocEnv(phenv);
    if (res != SQL_SUCCESS) {
        fprintf(stderr, "Unable to allocate environment handle (ret=%d)\n", res);
        exit(1);
    }

    // Allocate a connection handle
    res = SQLAllocConnect(*phenv, phdbc);
    if (res != SQL_SUCCESS) {
        fprintf(stderr, "Unable to allocate connection handle (ret=%d)\n", res);
        cleanup(*phenv, *phdbc, *phstmt);
        exit(1);
    }

    // The connection parameters
    printf("Connection Parameters: DSN='%s', UID='%s', PWD='%s'\n", DSN,
        DSN_userid, DSN_password);

    // Connect to the database
    res = SQLConnect(*phdbc, DSN, SQL_NTS, DSN_userid, SQL_NTS,
        DSN_password, SQL_NTS);
    if (res != SQL_SUCCESS) {
        fprintf(stderr, "Unable to connect to data source (ret=%d)\n", res);
        cleanup(*phenv, *phdbc, *phstmt);
        exit(1);
    }
}
```

```
// Allocate a statement handle
res = SQLAllocStmt(*phdbc, phstmt);
if (res != SQL_SUCCESS) {
    fprintf(stderr, "Unable to allocate statement handle (ret=%d)\n", res);
    cleanup(*phenv, *phdbc, *phstmt);
    exit(1);
}
printf("Handles initialized.\n");
return 0;
}

// Cleanup before exit
void cleanup(HENV henv, HDBC hdbc, HSTMT hstmt)
{
    // Disconnect from the database and free all handles
    SQLFreeStmt(hstmt, SQL_CLOSE);
    SQLDisconnect(hdbc);
    SQLFreeConnect(hdbc);
    SQLFreeEnv(henv);
    return;
}
```

PrepAndExec.c

```
#include <stdio.h>
#include <stdlib.h>
#include "iodbc/iodbc.h"
#include "iodbc/isqlext.h"
#include "Initialize.h"

int main(int argc, char *argv[])
{
    HENV henv;
    HDBC hdbc;
    HSTMT hstmt;
    SQLCHAR SelectStmt[255];
    int res;

    if (argc != 2) {
        fprintf(stderr, "Usage: PrepAndExec <SQL Statement>\n");
        exit(1);
    } else {
        if (strlen(argv[1]) > 254) {
            fprintf(stderr, "SQL statement is longer than 254 characters\n");
            exit(1);
        }
        strncpy((char *) SelectStmt, argv[1], 255);
    }

    // Allocate all handles
    res = Initialize(&henv, &hdbc, &hstmt);
    if (res != 0) {
        fprintf(stderr, "Unable to initialize (ret=%d)\n", res);
        exit(1);
    }

    // Prepare the user provided SQL statement
    res = SQLPrepare(hstmt, SelectStmt, SQL_NTS);
    if (res != SQL_SUCCESS) {
        fprintf(stderr, "Unable to prepare statement (ret=%d)\n", res);
        // Disconnect from the database and free all handles
        cleanup(henv, hdbc, hstmt);
        exit(1);
    }

    // Execute the SQL Statement
    res = SQLExecute(hstmt);
    if (res != SQL_SUCCESS) {
        fprintf(stderr, "Unable to execute statement (ret=%d)\n", res);
        // Disconnect from the database and free all handles
        cleanup(henv, hdbc, hstmt);
        exit(1);
    }
}
```

```
printf("'%s' prepared and executed.\n", SelectStmt);
cleanup(henv, hdbc, hstmt);
return 0;
}
```

ExecDirect.c

```
#include <stdio.h>
#include <stdlib.h>
#include "iodbc/iodbc.h"
#include "iodbc/isqlext.h"
#include "Initialize.h"

int main(int argc, char *argv[])
{
    HENV henv;
    HDBC hdbc;
    HSTMT hstmt;
    SQLCHAR SelectStmt[255];
    int res;

    if (argc != 2) {
        fprintf(stderr, "Usage: ExecDirect <SQL Statement>\n");
        exit(1);
    } else {
        if (strlen(argv[1]) > 254) {
            fprintf(stderr, "SQL statement is longer than 254 characters.\n");
            exit(1);
        }
        strncpy((char *) SelectStmt, argv[1], 255);
    }

    // Allocate all handles
    res = Initialize(&henv, &hdbc, &hstmt);
    if (res != 0) {
        fprintf(stderr, "Unable to initialize (ret=%d)\n", res);
        exit(1);
    }

    // Prepare and execute the user provided SQL statement
    res = SQLExecDirect(hstmt, SelectStmt, SQL_NTS);
    if (res != SQL_SUCCESS) {
        fprintf(stderr, "Unable to execute statement directly (ret=%d)\n", res);
        // Disconnect from the database and free all handles
        cleanup(henv, hdbc, hstmt);
        exit(1);
    }
    printf("'%'s' directly executed.\n", SelectStmt);
    cleanup(henv, hdbc, hstmt);
    return 0;
}
```

ViewTable.c

```
#include <stdio.h>
#include <stdlib.h>
#include "iodbc/iodbc.h"
#include "iodbc/isqlext.h"
#include "Initialize.h"

int main(int argc, char *argv[])
{
    HENV henv;
    HDBC hdbc;
    HSTMT hstmt;
    SQLCHAR SelectStmt[255], cname1[25], cname2[50];
    int res;

    if (argc != 1) {
        fprintf(stderr, "Usage: ViewTable\n");
        exit(1);
    }

    // Allocate all handles
    res = Initialize(&henv, &hdbc, &hstmt);
    if (res != 0) {
        fprintf(stderr, "Unable to initialize (ret=%d)\n", res);
        exit(1);
    }

    // Create the query
    strcpy((char *) SelectStmt, "SELECT * FROM players");

    // Prepare and execute the SQL statement
    res = SQLExecDirect(hstmt, SelectStmt, SQL_NTS);
    if (res != SQL_SUCCESS) {
        fprintf(stderr, "Unable to execute statement directly (ret=%d)\n", res);
        // Disconnect from the database and free all handles
        cleanup(henv, hdbc, hstmt);
        exit(1);
    }

    // Bind the columns in the result data set returned to
    // application variables
    res = SQLBindCol(hstmt, 1, SQL_C_CHAR, cname1, sizeof(cname1), NULL);
    if (res != SQL_SUCCESS) {
        fprintf(stderr, "Unable to bind column 1 (ret=%d)\n", res);
        cleanup(henv, hdbc, hstmt);
        exit(1);
    }
}
```

```
res = SQLBindCol(hstmt, 2, SQL_C_CHAR, cname2, sizeof(cname2), NULL);
if (res != SQL_SUCCESS) {
    fprintf(stderr, "Unable to bind column 2 (ret=%d)\n", res);
    cleanup(henv, hdbc, hstmt);
    exit(1);
}

printf("\nContent of players:\n");
printf("*****\n");

// While there are remaining rows in the result set,
// retrieve and display them
res = SQLFetch(hstmt);
while (res != SQL_NO_DATA_FOUND) {
    if (res != SQL_SUCCESS && res != SQL_SUCCESS_WITH_INFO) {
        fprintf(stderr, "Unable to fetch row (ret=%d)\n", res);
        cleanup(henv, hdbc, hstmt);
        exit(1);
    }
    printf("%s -- %s \n", cname1, cname2);
    res = SQLFetch(hstmt);
}
cleanup(henv, hdbc, hstmt);
return 0;
}
```

Transact.c

```
#include <stdio.h>
#include <stdlib.h>
#include "iodbc/iodbc.h"
#include "iodbc/isqlext.h"
#include "Initialize.h"

int main(int argc, char *argv[])
{
    HENV henv;
    HDBC hdbc;
    HSTMT hstmt;
    SQLCHAR SelectStmt[255];
    int res, commit = -1;

    if (argc != 2) {
        fprintf(stderr, "Usage: Transact <action>\n");
        fprintf(stderr, "action=1 to commit, action=0 to rollback");
        exit(1);
    } else {
        if (atoi(argv[1]) == 1) {
            commit = 1;
        } else if (atoi(argv[1]) == 0) {
            commit = 0;
        } else {
            fprintf(stderr, "Usage: Transact <action>\n");
            fprintf(stderr, "action=1 to commit, action=0 to rollback");
            exit(1);
        }
    }

    // Allocate all handles
    res = Initialize(&henv, &hdbc, &hstmt);
    if (res != 0) {
        fprintf(stderr, "Unable to initialize (ret=%d)\n", res);
        exit(1);
    }

    // Turn AUTOCOMMIT off so SQL statements can be grouped into a
    // transaction
    res = SQLSetConnectOption(hdbc, SQL_AUTOCOMMIT, SQL_AUTOCOMMIT_OFF);
    if (res != SQL_SUCCESS) {
        fprintf(stderr, "Unable to turn AUTOCOMMIT off (ret=%d)\n", res);
        cleanup(henv, hdbc, hstmt);
        exit(1);
    }
}
```

```
// Create update string
strcpy((char *) SelectStmt, "INSERT into players VALUES ");
strcat((char *) SelectStmt, "(\'Tracy McGrady\', \'Orlando Magic\')");

// Execute the SQL statement
res = SQLExecDirect(hstmt, SelectStmt, SQL_NTS);
if (res != SQL_SUCCESS) {
    fprintf(stderr, "Unable to directly execute statement (ret=%d)\n", res);
    // Disconnect from the database and free all handles
    cleanup(henv, hdbc, hstmt);
    exit(1);
}

// Create delete string
strcpy((char *) SelectStmt, "DELETE FROM players WHERE ");
strcat((char *) SelectStmt, "name=\'Michael Jordan\'");

// Execute the SQL statement
res = SQLExecDirect(hstmt, SelectStmt, SQL_NTS);
if (res != SQL_SUCCESS) {
    fprintf(stderr, "Unable to directly execute statement (ret=%d)\n", res);
    // Disconnect from the database and free all handles
    cleanup(henv, hdbc, hstmt);
    exit(1);
}

// Commit or rollback the transaction
if (commit == 1) {
    res = SQLTransact(henv, hdbc, SQL_COMMIT);
    printf("Transaction committed\n", res);
} else {
    res = SQLTransact(henv, hdbc, SQL_ROLLBACK);
    printf("Transaction rolled back\n", res);
}
if (res != SQL_SUCCESS) {
    fprintf(stderr, "Unable to commit the transaction (ret=%d)\n", res);
    cleanup(henv, hdbc, hstmt);
    exit(1);
}
cleanup(henv, hdbc, hstmt);
return 0;
}
```