Oracle/C++ Application Programming

C++ Call Interface Programmer's Guide

https://docs.oracle.com/database/121/LNCPP/relational.htm#LNCPP003

Agenda

- Connecting to a Database
- Executing SQL DDL and DML Statements
- Types of SQL Statements in the OCCI Environment
- Executing SQL Queries
- Committing a Transaction
- Handling Exceptions

Connecting to a Database

Creating and Terminating an Environment

- All OCCI processing takes place inside the Environment class.
- An OCCI environment provides application modes and user-specified memory management functions.
- To create an OCCI environment

```
Environment *env = Environment::createEnvironment();
```

- create*xxx*() methods are used to create OCCI objects such as
  - connections
  - Statements
- At the end of your program, you need to terminate the OCCI environment:

```
Environment::terminateEnvironment(env);
```

Opening and Closing a Connection

- The *Environment* class is the factory class for creating *Connection* objects.
- Before creating a connection, you need to create the environment.

- Use an environment instance to create a connection:

```
Environment *env = Environment::createEnvironment();
Connection *conn = env->createConnection("username", "password ", "Connection String");
```

- You must terminate a connection at the end of the session.

```
env->terminateConnection(conn);
Environment::terminateEnvironment(env);
```

Creating a Database Connection

```
1  #include <iostream>
2  #include <occi.h>
3
4  using oracle::occi::Environment;
5  using oracle::occi::Connection;
6  using namespace oracle::occi;
7  using namespace std;
8
9  int main(void)
10 {
11     /* OCCI Variables */
12     Environment* env = nullptr;
13     Connection* conn = nullptr;
14     /* Used Variables */
15     string str;
16     string user = "username";
17     string pass = "password";
18     string constr = "myoracle12c.senecacollege.ca:1521/oracle12c";
19     try {
20         env = Environment::createEnvironment(Environment::DEFAULT);
21         conn = env->createConnection(user, pass, constr);
22         cout << "Connection is Successful!" << endl;
23         env->terminateConnection(conn);
24         Environment::terminateEnvironment(env);
25     }
26     catch (SQLException& sqlExcp) {
27         cout << sqlExcp.getErrorCode() << ": " << sqlExcp.getMessage();
28     }
29     return 0;
30 }
```

Executing SQL DDL and DML Statements

Creating a Statement Object

- The *Statement* class is used to execute SQL commands.
- To create a statement objects:

```
Statement *stmt = conn->createStatement();
```

Execute SQL Commands

- After creating the statement object, the following methods can be called to execute SQL commands:
  - execute()
    - executes all nonspecific statement types
  - executeUpdate()
    - executes DML and DDL statements
  - executeArrayUpdate()
    - executes multiple DML statements
  - executeQuery()
    - executes a query
- To creates a table in a database:

```
stmt->executeUpdate("CREATE TABLE student
(s_id NUMBER(4), name VARCHAR2(40))");
```

- To insert values into a table

```
stmt->executeUpdate("INSERT INTO student
VALUES(10, 'Sarah Stone')");
```

executeUpdate()

- The *executeUpdate()* executes a SQL INSERT, UPDATE, DELETE, and a DDL statements CREATE/ALTER.
- It returns the number of rows affected by the SQL statement execution.

Terminating a Statement Object

- Terminate and deallocate a *Statement* object using the following statement:
  - terminateStatement()
- See the following code that closes a statement object stmt:

```
Connection::conn->terminateStatement(Statement *stmt);
```

executeUpdate() – Create a Table

```
try {

    env = Environment::createEnvironment(Environment::DEFAULT);
    conn = env->createConnection(user, pass, constr);

    stmt = conn->createStatement("CREATE TABLE student (s_id NUMBER(4), name VARCHAR2(40))");
    stmt->executeUpdate();

    cout << "Table Created successfully." << endl;

    conn->terminateStatement(stmt);
    env->terminateConnection(conn);
    Environment::terminateEnvironment(env);
}
catch (SQLException& sqlExcp) {
    cout << "error";
    cout << sqlExcp.getErrorCode() << ": " << sqlExcp.getMessage();
}
```

executeUpdate() – Drop a Table

```
try {

    env = Environment::createEnvironment(Environment::DEFAULT);
    conn = env->createConnection(user, pass, constr);

    stmt = conn->createStatement("drop table student");
    stmt->executeUpdate();

    cout << "Table deleted successfully." << endl;

    conn->terminateStatement(stmt);
    env->terminateConnection(conn);
    Environment::terminateEnvironment(env);
}
catch (SQLException& sqlExcp) {
    cout << "error";
    cout << sqlExcp.getErrorCode() << ": " << sqlExcp.getMessage();
}
```

createStatement - Example

```
stmt = conn->createStatement("CREATE TABLE student (s_id NUMBER(4), name VARCHAR2(40))");
stmt->executeUpdate();
```

or

```
stmt = conn->createStatement();
stmt->executeUpdate("CREATE TABLE student (s_id NUMBER(4), name VARCHAR2(40))");
```

Types of SQL Statements in the OCCI Environment

Oracle® C++ Call Interface

Programmer's Guide

12*c* Release 1 (12.1)

E48221-07

Types of SQL Statements

- There are three types of SQL statements in OCCI.
  - Standard Statements
    - use SQL commands with specified values
  - Parameterized Statements
    - have parameters, or bind variables
  - Callable Statements
    - call stored PL/SQL procedures and functions

Standard Statements

- In standard statements, the values are explicitly specified.
- See the following example:
  - To creates a table in a database:

```
stmt->executeUpdate("CREATE TABLE student
(s_id NUMBER(4), name VARCHAR2(40))");
```

  - To insert values into a table

```
stmt->executeUpdate("INSERT INTO student
VALUES(10, 'Sarah Stone')");
```

Standard Insert Statement

```
try {

    env = Environment::createEnvironment(Environment::DEFAULT);
    conn = env->createConnection(user, pass, constr);

    stmt = conn->createStatement("INSERT INTO student VALUES (10, 'Sarah Stone')");
    stmt->executeUpdate();

    cout << "Row Inserted successfully." << endl;

    conn->terminateStatement(stmt);
    env->terminateConnection(conn);
    Environment::terminateEnvironment(env);
}
catch (SQLException& sqlExcp) {
    cout << "error";
    cout << sqlExcp.getErrorCode() << ": " << sqlExcp.getMessage();
}
```

Parameterized Statements

- A statement can be executed with different values using parameters as placeholders for the input values.
  - The *setxxx()* is used to specify parameters.
  - *XXX* stands for the type of the parameter.
- See the following example:

```
setSQL("INSERT INTO student VALUES (:1,:2) ");
```

  - You first need to specify the statement using the *setSQL()* method.

```
stmt->setInt(1, 1003); // value for first parameter
stmt->setString(2, "Nick Shine"); // value for second parameter
stmt->executeUpdate(); // execute statement
...
stmt->setInt(1, 1004); // value for first parameter
stmt->setString(2, "Adam Sandler"); // value for second parameter
stmt->executeUpdate(); // execute statement
```

setSQL( )

- The setSQL() method is used to reuse a statement object to store and execute a SQL statement multiple times.

```
setSQL("INSERT INTO student VALUES (:1,:2) ");
```

- The getSQL() method can be called to the content of the current statement.
- To reset a statement object, call serSQL() method with the new SQL statement.

```
stmt->setSQL("SELECT * FROM inventories WHERE quantity < :1");
```

Parameterized Insert Statement

```
try {

    env = Environment::createEnvironment(Environment::DEFAULT);
    conn = env->createConnection(user, pass, constr);

    stmt = conn->createStatement();
    stmt->setSQL("INSERT INTO student VALUES (:1,:2) ");
    // insert one row
    stmt->setInt(1, 1003); // value for first parameter
    stmt->setString(2, "Nick Shine"); // value for second parameter
    stmt->executeUpdate();

    cout << "First Student Inserted Successfully." << endl;

    //insert another row
    stmt->setInt(1, 1004); // value for first parameter
    stmt->setString(2, "Adam Sandler"); // value for second parameter
    stmt->executeUpdate();

    cout << "Second Student Inserted Successfully." << endl;

    conn->terminateStatement(stmt);
    env->terminateConnection(conn);
    Environment::terminateEnvironment(env);
}
```

Reset Statement Objects

The method setSQL() can b used to reset a statement object to be used for a different SQL statement.

```
try {

    env = Environment::createEnvironment(Environment::DEFAULT);
    conn = env->createConnection(user, pass, constr);

    stmt = conn->createStatement();
    stmt->setSQL("INSERT INTO student VALUES (:1,:2) ");
    // insert one row
    stmt->setInt(1, 1005); // value for first parameter
    stmt->setString(2, "Tim Black"); // value for second parameter
    stmt->executeUpdate();

    cout << "Student Inserted Successfully." << endl;

    //reset the statement
    stmt->setSQL("DELETE FROM student WHERE s_id = :1");
    stmt->setInt(1, 1004); // value for first parameter
    stmt->executeUpdate();

    cout << "Student Deleted Successfully." << endl;

    conn->terminateStatement(stmt);
    env->terminateConnection(conn);
    Environment::terminateEnvironment(env);
}
```

Callable Statements

- PL/SQL stored procedures are procedures stored on a database server which can be called inside a database or by an application.
- First define the statement to be executed:

```
stmt->setSQL("BEGIN countStudents(:1, :2); END:");
```

  - The above command will call the *coutStudents* stored procedure that has two parameters.
  - First parameter is an *IN* parameter. It gets the value of PGM (program). The stored procedure will then find the number of students in the "CPA" program.

```
stmt->setString(1, "CPA");
```

  - The second parameter is an *OUT* parameter. It stored the number of students in the "CPA" program and the values will be returned to the caller.

```
int count;  // this variable stores the returning value from the
countStudent() procedure

stmt->registerOutParam(2, Type::OCCIINT, sizeof(count));
// specify type and size of the second (OUT) parameter
```

  - And finally, execute the statement to call and execute the stored procedure.

```
stmt->executeUpdate(); // call the procedure
```

  - Now, save the returning value of the OUT parameter in the count variable.

```
count = stmt->getInt(2);
```

Executing SQL Queries

- An application fetch information from a database by executing SQL queries.
- You can execute a query and store a result into a result set object.
- See the following example:

```
ResultSet *rs = stmt->executeQuery("SELECT * FROM student");
```

  - The result of the above query is stored in *rs*.

```
cout << "The CPA program has:" << endl;
while (rs->next())
{
int count = rs->getInt(1); // get the first column as int
string name = rs->getString(2); // get the second column as string
cout << count << " " << students << endl;
}
```

  - You can perform operations on data in the result set.
  - The next() method is used to fetch the next row.
  - The getXXX() is used to fetch the value of a column.

ResultSet Class

- A ResultSet provides access to the result of a query.
- It provides a cursor pointing to the current row.
- The cursor initially is pointing to the position before the first row.
- The *next()* method moves the cursor to the next row.
- The *getxxx()* method is used to fetch the value of a column.

```
ResultSet::getxxx()
while (rs->next())
{
int count = rs->getInt(1); // get the first column as int
string name = rs->getString(2); // get the second column as string
cout << count << " " << students << endl;
}
```

Query with Variables

- Variables can be used to specify values in the WHERE clause of a SQL query.
- We want to find students with gpa 3.2:

```
stmt->setSQL("SELECT * FROM student WHERE gpa >= :1");
float gpa = 3.2;
stmt->setFloat(1, gpa); // set the parameter
ResultSet *rs = stmt->executeQuery();
cout << "The students with GPA 3.2:" << endl;
while (rs->next())
      cout << rs->getInt(1) << " " << rs->getString(2) << endl;
```

Standard SQL Query

```
try {

    env = Environment::createEnvironment(Environment::DEFAULT);
    conn = env->createConnection(user, pass, constr);

    Statement* stmt = conn->createStatement("SELECT * FROM student");
    ResultSet* rs = stmt->executeQuery();

    if (!rs->next()) {
        // if the result set is empty
        cout << "ResultSet is empty." << endl;
    }
    else {
        // if the result set in not empty
        do {
            cout << "Student ID: " << rs->getInt(1) << " Student Name: " << rs->getString(2) << endl;
        } while (rs->next()); //if there is more rows, iterate
    }

    conn->terminateStatement(stmt);
    env->terminateConnection(conn);
    Environment::terminateEnvironment(env);
}
```

SQL Query with Parameters

```
try {

    env = Environment::createEnvironment(Environment::DEFAULT);
    conn = env->createConnection(user, pass, constr);

    Statement* stmt = conn->createStatement("SELECT * FROM student WHERE s_id = :1");
    stmt->setInt(1,1004);
    ResultSet* rs = stmt->executeQuery();

    if (!rs->next()) {
        // if the result set is empty
        cout << "ResultSet is empty." << endl;
    }
    else {
        // if the result set in not empty
        do {
            cout << "Student ID: " << rs->getInt(1) << " Student Name: " << rs->getString(2) << endl;
        } while (rs->next()); //if there is more rows, iterate
    }
```

getXXX() Methods

Method	Description
getDate()	Return the value of a parameter as a Date object
getDouble()	Return the value of a parameter as a C++ double.
getFloat()	Return the value of a parameter as a C++ float.
getInt()	Return the value of a parameter as a C++ int.
getNumber()	Return the value of a parameter as a Number object.
getString()	Return the value of the parameter as a string.

isNull() Method

- Checks whether the parameter is null.

```
Statement* stmt = conn->createStatement("SELECT * FROM student WHERE s_id = :1");
stmt->setInt(1,1005);
ResultSet* rs = stmt->executeQuery();

if (!rs->next()) {
    // if the result set is empty
    cout << "ResultSet is empty." << endl;
}
else {
    // if the result set in not empty
    do {
        if (!rs->isNull(2)) //if the column name is not null
        {
            cout << "Student ID: " << rs->getInt(1) << " Student Name: " << rs->getString(2) << endl;
        } else
        {
            cout << "Student ID: " << rs->getInt(1) << " Student Name: " << "Unknown" << endl;
        }
    } while (rs->next()); //if there is more rows, iterate
}
```

setXXX() Methods

Method	Description
setDate()	Set a parameter to a Date value.
setDouble()	Set a parameter to a C++ double value.
setFloat()	Set a parameter to a C++ float value.
setInt()	Set a parameter to a C++ int value.
setNull()	Set a parameter to SQL null.
setNumber()	Set a parameter to a Number value.
setString()	Set a parameter to an string value.

Transactions

- Changes by DDL statements become permanent after committing the transaction or reversed by rollback.
- Commit and Rollback commands can be execute when using executeUpdate().
  - `Connection::commit()`
  - `Connection::rollback()`
- To make changes of DML statements permanent immediately, execute the following command:

```
Statement::setAutoCommit(TRUE);
```

- To set the auto commit off:

```
Statement::setAutoCommit(FALSE);
```

Handling Exceptions

- OCCI methods generates exceptions of type *SQLException* if they are unsuccessful.
  - The SQLException class contains Oracle specific error numbers and messages.
  - The error message can be obtained by the *exception::what()* or getMessage() method.
  - The getErrorCode() returns the Orcale error code.
- See the following examples:

```
catch (exception &excp)
{
cerr << excp.what() <<
endl;
}
```

```
catch (SQLException &sqlExcp)
{
cerr <<sqlExcp.getErrorCode << ": " <<
sqlExcp.getErrorMessage() << endl;
}
catch (exception &excp)
{
cerr << excp.what() << endl;
}
```