The Oracle SQL Class Library

Version 1.0

Getting Started

Sergei Kuchin

Mosakin Corporation
1070-R Highway 34
Matawan, NJ 07747-3459

Email: kuchin@empros.com
kuchin@mosakin.com

1995

Copyright (C) Mosakin Corp., 1995

Permission to use, copy, modify and redistribute
this document and the Oracle SQL class library for
any purpose is hereby granted without fee,
provided that the above copyright notice appear in
all copies.

TABLE OF CONTENTS

1. Introduction 4

2. Streams and SQL 4
2.1. Example 5

3. The SQL pipe 8
3.1. Example in C++ 8
3.2. Example in C 9

4. Macros for hard-coding SQL into a program 11
4.1. Example 11

5. Exception handling 13
5.1. Example 13

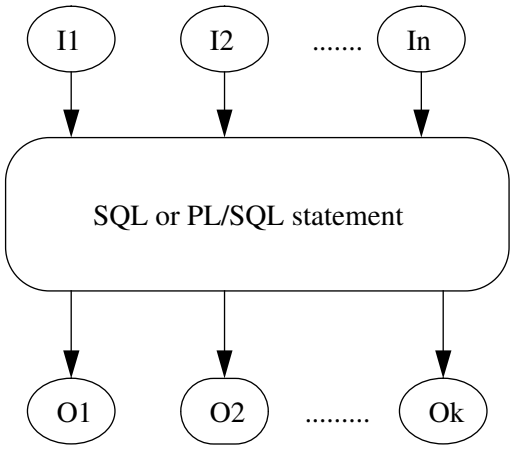
1. Introduction

This document is an introduction into the Oracle SQL class library (OSCL). The library is a collection of C++ classes and macros. This document is organized in the following manner:

- Stream classes for SQL and PL/SQL operations (Chapter 2)
- The SQL pipe and the SQL pipe requestors for C and C++ (Chapter 3)
- Macros for hard-coding SQL and PL/SQL statements into a database application (Chapter 4)
- Exception handling (Chapter 5)
- low level interface classes(C++/Oracle Call Interface driver); for more details refer Oracle SQL Class Library Programmer's Guide;

2. Streams and SQL

In the Oracle SQL class library SQL streams are introduced. The idea here, is to combine streams and SQL giving a new quality and simplicity to the programming interface to SQL. The Oracle array interface naturally transforms into buffered stream operations. Any SQL (or PL/SQL) statement can be treated as a functional element with input/output variables:



A SQL statement is executed when values of all its input variables are defined. Resulting values are then assigned into the output variables immediately after the execution.

Logically, a SQL stream has input and output formats. The input format for the SQL stream is defined by a set of output variables of the associated SQL statement. Similarly, the output format for SQL stream is defined by input variables of the corresponding SQL statement. When objects are written into the stream, values are actually assigned to the input variables. Likewise, when objects are read from the stream, values are read from the output variable of the SQL statement.

The most convenient class amongst the stream classes is the SQL stream class. The SQL stream is similar to buffered file streams. A SQL statement is opened as an ordinary file. The SQL stream has also a flush function for flushing buffers and a collection of >> and << operators for reading and writing objects of different data types. In addition, the SQL stream is unified for SQL or PL/SQL statements of any kind.

The SQL stream has a small parser for parsing declarations of bind variables and their data types. There is no need to declare C/C++ host variables and bind them with placeholders by special bind functions calls. All the host variables are created dynamically inside the library functions. A stream just needs to be opened for writing input values into it and reading output values.

2.1. Example

Let's assume you have a table which contains information about states: state code, state name and state capital name. Suppose, you want to insert a few rows into the table. This can be accomplished code:

```
SQLStream o(db, // "Connect" object
        "insert into state values( "
        "     :state<char[33]>, "
        "     :capital<char[20]>, "
        "     :state_code<int>)",
        5 // Internal buffer size.
          // The INSERT statement will be executed
          // for 5 rows per operation.
   );

char state[33];
char capital[20];
int state_code;

while(!cin.eof()){
   cin>>state>>capital>>state_code;
   o<<state<<capital<<state_code;
}
```

Note that the stream is buffered and the INSERT is executed (i.e. the stream is flushed) when the internal buffer gets full.

By default, the transaction is committed when the stream is flushed. If you want to postpone committing the transaction to the end, unset the "Commit" flag:

```
o.SetCommit(0);

while(!cin.eof()){
  cin>>state>>capital>>state_code;
  o<<state<<capital<<state_code;
}
```

```
o.commit(); // Flush the buffer and commit transaction
```

Let's say you want to select state names and state capitals for state codes less than 40 and for all states which have the first letter "M" in their names. The following code does the trick.

```
SQLStream in(db, // "Connect" object

        "select state, capital "
        "from state "
        "where "
        "  state_code <= :v_code<int> and "
        "  state like :v_state<char[33]> "
        "order by state_code",

        10 // Internal buffer size,
          // i.e. output rows will be fetched
          // by 10 rows per one fetch operation.
          // Increasing the buffer size you can
          // boost the program performance.
      );

char state[33];
capital char[20];

in<<40<<"M%";

while(!in.eof()){
   in>>state>>capital;
   cout<<state<<","<<capital<<endl;
}


// Now you  want to enter  new input values  and re-execute the
// query

  in<<20<<"C%";

//  When input  values  are  assigned, the SQL   statement is
// automatically re-executed

  while(!in.eof()){
     in>>state>>capital;
     cout<<state<<capital<<endl;
  }
```

Full text of the example is given below.

```
#include <iostream.h>
#include <oscl.h>

 char state[33];
 char capital[20];
 int state_code;
```

```
void Insert(SQL* db)
{
 SQLStream o(db, // “Connect” object
            “insert into state values( “
            “      :state<char[33]>, “
            “      :capital<char[20]>, “
            “      :state_code<int>)”,
            5 // Internal buffer size.
              // The INSERT statement will be executed
              // for 5 rows per operation.
            );

 while(!cin.eof()){
    cin>>state>>capital>>state_code;
    o<<state<<capital<<state_code;
 }
} /* Insert */

void Select(SQL* db)
{
 SQLStream in(db, // “Connect” object
             “select state, capital “
             “from state “
             “where “
             “  state_code <= :v_code<int> and “
             “  state like :v_state<char[33]> “
             “order by state_code”,
             10 // Internal buffer size,
                // i.e. output rows will be fetched
                // by 10 rows per one fetch operation.
                // Increasing the buffer size you can
               // boost the program performance.
             );

 in<<40<<“M%”;

 while(!in.eof()){
    in>>state>>capital;
    cout<<state<<“,”<<capital<<endl;
 }

} /* Select */
int main()
{
 SQL db(“scott/tiger”); // Connect to Oracle
 Insert(&db);           // Input new records
 Select(&db);           // Select records
 return 0;
} /* main */
```

3. The SQL pipe

The SQL pipe is a stand alone program which encapsulates all Oracle runtime libraries inside one Unix application. It can be run by a very small client program. The client program must be linked with a SQL pipe requestor which has a programming interface to the SQL pipe. The SQL pipe, in its turn, sends over all requests from the client to the Oracle Server.

Separate SQL pipe requestors for C and C++ are included in the source code.

The SQL pipe requestor for C++ is almost identical to the SQL stream interface. The SQL pipe requestor for C has scanf/printf-like functions which makes it fairly easy handle.

The main advantage of using this SQL pipe requestors is their simplicity and their built-in transparent Oracle array interface. In addition, it obviates the need to link Oracle libraries into the application.

A client program runs an instance of the SQL pipe program, connects to Oracle, performs SQL or PL/SQL calls and then disconnects. When an Oracle error occurs, the SQL pipe terminates by sending an error message to the client. One option is to to execute a constant SQL statement with the “ignore error” flag set.

3.1. Example in C++

This is the example from 2.1. rewritten for the SQL pipe requestor in C++:

```
#include <iostream.h>
#include <sql_req.h>

 char state[33];
 char capital[20];
 int state_code;

void my_handler(char* err_msg)
{
 cout<<err_msg<<endl;
}

void Insert()
{
 SQLStream o(
          “insert into state values( “
          “     :state<char[33]>, “
          “     :capital<char[20]>, “
          “     :state_code<int>)”,

          5 // Internal buffer size.
         );
```

```
 while(!cin.eof()){
   cin>>state>>capital>>state_code;
   o<<state<<capital<<state_code;
 }
} /* Insert */

void Select()
{
 SQLStream in(
            "select state, capital "
            "from state "
            "where "
            "  state_code <= :v_code<int> and "
            "  state like :v_state<char[33]> "
            "order by state_code",

            10 // Internal buffer size,
            );

 in<<40<<"M%";

 while(!in.eof()){
   in>>state>>capital;
   cout<<state<<","<<capital<<endl;
 }

} /* Select */

int main()
{
 SQL::set_handler(my_handler);
 cout<<"open="<<SQL::open("scott/tiger")<<endl;

 Insert();
 Select();

 SQL::close();

 return 0;
} /* main */
```

3.2. Example in C

This is the example from 2.1. rewritten for the SQL pipe requestor in C:

```
#include <stdio.h>
#include <csql_req.h>

 char state[33];
 char capital[20];
 int state_code;
```

```
void my_handler(char* err_msg)
{
 printf("%s\n",err_msg);
}

void Insert()
{
 int o=sql_stream_open(
            "insert into state values( "
            "      :state<char[33]>, "
            "      :capital<char[20]>, "
            "      :state_code<int>)",
            5
       );

 while(!feof(stdin)){
    fscanf(stdin,"%s %s %d",state,capital,&state_code);
    sql_printf(o,"%s %s %d",state,capital,state_code);
 }
 sql_stream_close(o);
} /* Insert */

void Select()
{int in;
 in=sql_stream_open(
             "select state, capital "
             "from state "
             "where "
             "  state_code <= :v_code<int> and "
             "  state like :v_state<char[33]> "
             "order by state_code",
             10
   );

 sql_printf(in,"%d %s",40,"M%");

 while(!sql_eof(in)){
    sql_scanf(in,"%s %s",state,capital);
    printf("%s,%s\n",state,capital);
 }
 sql_stream_close(in);
} /* Select */

int main()
{
 sql_set_handler(my_handler);
 printf("open=%d\n",sql_open("scott/tiger"));

 Insert();
 Select();

 sql_close();

 return 0;
```

```
} /* main */
```

4. Macros for hard-coding SQL into a program

The library provides a collection of C++ macros for hard-coding SQL statements into source code. This style of embedding SQL statements is similar to Embedded SQL in Pro*C but has the added advantage of not requiring any special precompilers.

4.1. Example

```
#include <iostream.h>
#include <string.h>
#include <oscl.h>

 PConnect db; // “Connect” object

class InsertState{
public:

 STRING_VAR(state,33);   // Define host variable “state”
 STRING_VAR(capital,20); // Define host variable “capital”
 INT_VAR(state_code);    // Define host variable “state_code”

 BEGIN_VAR_LIST(istm)  // Define host variable list for “istm”
   VAR(state),
   VAR(capital),
   VAR(state_code),
 END_VAR_LIST;

 DEFINE_EMPTY_SELECT_LIST(istm); // Define empty select list for “istm”

 DEFINE_STM(istm, // Declare SQL statement “istm”
          “insert into state values( “
          “      :state, “
          “      :capital, “
          “      :state_code)”
         );


 InsertState(PConnect db){istmOpen(db);} // Open “istm”
 ~InsertState(){istmClose();} // Close “istm”

};

void Insert(PConnect db)
{
 InsertState o(db);

 while(!cin.eof()){
    cin>>o.state>>o.capital>>o.state_code;
    o.istm->Exec();
```

```
 }
} /* Insert */


class SelectState{
public:

 STRING_ITEM(state,33);    // Define select list item “state”
 STRING_ITEM(capital,20);  // Define select list item “capital”

 STRING_VAR(v_state,33);   // Define host variable “v_state”
 INT_VAR(v_code);          // Define host variable “v_code”

 BEGIN_SELECT_LIST(stm) // Define select list for “stm”
   ITEM(state),
   ITEM(capital),
 END_SELECT_LIST;

 BEGIN_VAR_LIST(stm) // Define host variable list for “stm”
   VAR(v_state),
   VAR(v_code),
 END_VAR_LIST;

 DEFINE_SELECT_STM(stm, // Declare SELECT statement “stm”
             “select state, capital “
             “from state “
             “where “
             “  state_code <= :v_code and “
             “  state like :v_state “
             “order by state_code”,
            10 // Internal buffer size
           );

 SelectState(PConnect db){stmOpen(db);} // Open “stm”
~SelectState(){stmClose();} // Close “stm”

};

void Select(PConnect db)
{
 SelectState in(db);

 in.v_code=40; // Assign 40 to v_code
 strcpy(in.v_state,”M%”); // Assign “M%” to v_state

 int rc;
 do{
    rc=in.stm->GetNext(); // Fetch next row
    if(rc==FetchOk)
     cout<<in.state<<“,”<<in.capital<<endl; // Print the row
 }while(rc==FetchOk);

} /* Select */
```

```
int main()
{
  TConnect db("scott/tiger"); // Connect to Oracle
  Insert(&db);              // Input new records
  Select(&db);               // Select records
  return 0;
} /* main */
```

5. Exception handling

The OSCL library has an optional exception handling mechanism. The library functions can be allowed to raise exceptions when Oracle errors occur. These exceptions can then be intercepted and handled.

The exception handling for different C++ compilers has been implemented differently.

For C++ compilers which support exceptions, the mechanism was implemented on top of the regular C++ exceptions via the oraException class.

For C++ compilers which do not support exceptions, the exception handling was implemented by means of long jumps. This implementation imposes some restrictions. Nested try and catch blocks cannot be used. Also, it is not permissible to place C++ objects on a program stack in order to prevent the program crash.

All the implementation details are hidden in the exception handling macros that should be used to make source code portable across different C++ compilers. Macro calls remain the same regardless of the macro expansions.

5.1. Example

This is the example from 2.1. Exception handling macro calls have been added to the program.

```
#include <iostream.h>
#include <oscl.h>

 SQL* db=0; // "Connect" object

 char state[33];
 char capital[20];
 int state_code;

void Insert(SQL* db)
{
 SQLStream o(db, // "Connect" object

          "insert into state values( "
          "      :state<char[33]>, "
```

```
            "     :capital<char[20]>, "
            "     :state_code<int>)",

            5 // Internal buffer size.
              // The INSERT statement will be executed
              // for 5 rows per operation.

            );

 while(!cin.eof()){
  cin>>state>>capital>>state_code;
  o<<state<<capital<<state_code;
 }
} /* Insert */

void Select(SQL* db)
{
 SQLStream in(db, // "Connect" object

             "select state, capital "
             "from state "
             "where "
             "  state_code <= :v_code<int> and "
             "  state like :v_state<char[33]> "
             "order by state_code",

             10 // Internal buffer size,
                // i.e. output rows will be fetched
                // by 10 rows per one fetch operation.
                // Increasing the buffer size you can
               // boost the program performance.
             );

 in<<40<<"M%";

 while(!in.eof()){
  in>>state>>capital;
  cout<<state<<","<<capital<<endl;
 }

} /* Select */

int main()
{
 EXCEPTION_ENABLE; // Make exceptions enable
 BEGIN_TRY        // Begin try block
  db=new SQL("scott/tiger"); // Connect to Oracle
  Insert(db);                // Input new records
  Select(db);                // Select records

  delete db; // Disconnect from Oracle
  return 0;
 END_TRY     // End try block
 BEGIN_CATCH // Begin catch block
```

```
 cout<<SQL::err_msg()<<endl;  // Print error message
EPILOG                        // Start EPILOG section
 delete db;                   // Here you may delete objects
                              // allocated in dynamic memory
 return 1;
END_CATCH                 // End catch block
} /* main */
```