

# Application Programming and SQL: ODBC

## Spring 2018

School of Computer Science  
University of Waterloo

Databases CS348

# Call Level Interface/ODBC

An interface built on a library calls:

- Applications are developed without access to the DB (and without additional tools: no precompilation)
- incorporates ODBC (MS) and X/Open standards
- but it is harder to use and **doesn't allow preprocessing** (e.g., no checking of your SQL code and data types)

Three fundamental objects in an ODBC program:

- Environments
- Connections
- Statements

# Connect and Disconnect

```
int main()
{
    SQLHENV    henv;
    SQLHDBC    hdbc;
    SQLRETURN  rc;
    SQLCHAR    server[SQL_MAX_DSN_LENGTH + 1] = "DBCLASS";
    SQLCHAR    uid[19] = "<your uid>";
    SQLCHAR    pwd[31] = "<your password>";

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

    rc = SQLConnect(hdbc, server, SQL_NTS, uid, SQL_NTS, pwd, SQL_NTS);
    if (rc != SQL_SUCCESS) {
        printf("Error connecting to %s\n", server); exit(1);
    } else printf("Connected to %s\n", server);

    /* DO SOMETHING HERE */

    SQLDisconnect(hdbc);
    SQLFreeConnect(hdbc);
    SQLFreeEnv(henv);
}
```

# Errors

- `SQLxxx` functions return error codes
  - ⇒ similar to `libc` functions
  - ⇒ we should check them after every `SQLxxx` call
- the actual return codes:
  - `SQL_SUCCESS`
  - `SQL_ERROR`
- use the `SQLERROR` function to get sensible messages

# SQL Statements

... and what we can do with them:

- `SQLAllocStmt` (allocates object)
- `SQLExecDirect` (execute)
- `SQLPrepare` (compile statement)
- `SQLExecute` (execute compiled statement)
- `SQLSetParam` (initialize a procedure parameter)
- `SQLNumResultCols` (number of result columns)
- `SQLBindCol` ("host variables" in ODBC)
- `SQLGetData` (obtaining values of result columns)
- `SQLFetch` (cursor access in ODBC)
- `SQLError` (obtains diagnostics)
- `SQLRowCount` (number of affected rows)
- ...
- `SQLFreeStmt` (frees object)

# Parameters

## 1 parameter markers

'?' in the text of the query

`SQLNumParams`

`SQLBindParameter`

## 2 results of queries

⇒ specified by the number of resulting columns

`SQLNumResultsCol`

`SQLDescribeCol`

`SQLBindCol` or `SQLGetData`

## 3 number of affected tuples (updates):

`SQLRowCount`

# Parameter Markers

The parameters are **bound** to buffers

```
SQLBindParameter(stmt-handle,  
                 param-nr,  
                 inp/out,  
                 c-type,  
                 db-type,  
                 db-prec,  
                 db-scale,  
                 val-ptr, val-len,  
                 val-NULL-ptr)
```

- ⇒ it substitutes a value pointed to by `val-ptr` (with length `val-len`, indicator variable `val-NULL-ptr`, and C data type `c-type`)
- ⇒ for the `param-nr`-th parameter of `stmt-handle`  
(using database type `db-type`).

# Example

```
SQLCHAR  stmt[] = "UPDATE author SET url = ? WHERE aid = ?";

SQLINTEGER aid;
SQLCHAR    s[70];
SQLINTEGER ind;

rc = SQLAllocStmt(hdbc, &hstmt);

rc = SQLPrepare(hstmt, stmt, SQL_NTS);

printf("Enter Author ID:  "); scanf("%ld",&aid);
printf("Enter Author URL: "); scanf("%s", s);

rc = SQLBindParameter(hstmt, 1,
                      SQL_PARAM_INPUT, SQL_C_CHAR,
                      SQL_CHAR, 0, 0, s, 70, &ind);

rc = SQLBindParameter(hstmt, 2,
                      SQL_PARAM_INPUT, SQL_C_SLONG,
                      SQL_INTEGER, 0, 0, &aid, 0, NULL);

rc = SQLExecute(hstmt);
```



# Late Binding of Parameters

- executing a statement without binding all parameters

⇒ more precisely, the last arg of `SQLBindParam`

points to the constant `SQL_DATA_AT_EXEC`

gives `SQL_NEED_DATA` error code.

- we can supply the parameters late using the following two functions (as many times as needed):
  - `SQLParamData` find which argument
  - `SQLPutData` supply data
- allows updating several rows using one update statement, but supplying different values for each individual row to be updated.

## How to get output values from a statement

- number of affected: `SQLRowCount`
- answers to queries:
  - 1 bind variables before execution: `SQLBindCol`
  - 2 get values after execution: `SQLGetData`
- get next tuple: `SQLFetch`

the result of `SQLFetch` is just a result code!

# A Query with SQLBindCol

```
SQLCHAR  sqlstmt[] = "SELECT pubid, title FROM publication";

SQLINTEGER  rows;
struct { SQLINTEGER ind;
        SQLCHAR    s[70];
        } pubid, title;

rc = SQLAllocStmt(hdbc, &hstmt);

rc = SQLExecDirect(hstmt, sqlstmt, SQL_NTS);

rc = SQLBindCol(hstmt, 1, SQL_C_CHAR,
               (SQLPOINTER)pubid.s, 8, &pubid.ind);
rc = SQLBindCol(hstmt, 2, SQL_C_CHAR,
               (SQLPOINTER)title.s, 70, &title.ind);

while ((rc = SQLFetch(hstmt)) == SQL_SUCCESS)
    printf("%-8.8s %-70.70s\n", pubid.s, title.s);

rc = SQLRowCount(hstmt, &rows);
printf(" %d rows selected\n", rows);

rc = SQLFreeStmt(hstmt, SQL_DROP);
```

# A Query with SQLGetData

```
SQLCHAR  sqlstmt[] = "SELECT pubid, title FROM publication";

SQLINTEGER  rows;
struct { SQLINTEGER ind;
        SQLCHAR      s[70];
        } pubid, title;

rc = SQLAllocStmt(hdbc, &hstmt);

rc = SQLExecDirect(hstmt, sqlstmt, SQL_NTS);

while ((rc = SQLFetch(hstmt)) == SQL_SUCCESS) {
    rc = SQLGetData(hstmt, 1, SQL_C_CHAR,
        (SQLPOINTER) pubid.s, 8, &(pubid.ind));
    rc = SQLGetData(hstmt, 2, SQL_C_CHAR,
        (SQLPOINTER) title.s, 70, &(title.ind));
    printf("%-8.8s %-70.70s \n", pubid.s, title.s);
}

rc = SQLRowCount(hstmt, &rows);
printf(" %d rows selected\n", rows);

rc = SQLFreeStmt(hstmt, SQL_DROP);
```

# Describe Columns

If we don't know number of columns/type/name, ...

```
SQLNumResultCols(hstmt, &num)
SQLDescribeCol(hstmt, ColNo,
               ColNamebuf, sizeof(ColNamebuf),
               NULL,
               &sqltype, &sqlprec, &sqlscale,
               &ifNullable );
```

```
SQLINTEGER sqlprec;
SQLSMALLINT i, num, sqltype, sqlscale, nullable;
SQLCHAR name[32];
```

```
rc = SQLNumResultCols(hstmt, &num);
```

```
for (i=0; i<num; i++) {
    rc = SQLDescribeCol(hstmt, i+1, name, 32, NULL,
                       &sqltype, &sqlprec, &sqlscale, &nullable);
    printf("attribute %d is %s (%d,%ld,%d,%d)\n"
           i, name, sqltype, sqlprec, sqlscale, nullable);
}
```

# Transactions

- transaction start:

⇒ implicitly using one of

SQLPrepare,  
SQLExecute,  
SQLExecDirect, etc.

functions.

- transaction end:

SQLTransact(henv, hdbc, what)

where

what = SQL\_COMMIT, or  
what = SQL\_ROLLBACK

# Summary

- CLI/ODBC can do everything Embedded SQL can.
- However, all statements are *dynamic*
  - ⇒ no precompilation
  - ⇒ explicit binding of parameters (user has to make types match!)
- An almost standard (ODBC, X/Open)
  - ⇒ independence on DBMS
  - ⇒ but: the standard has 100's of functions