CLI and ODBC

University of Waterloo

List of Slides

1
2 Call Level Interface/ODBC
3 ODBC Application Structure
4 Connect and Disconnect
5 Errors
6 SQL Statements
7 Statement Life
8 Parameters
9 Parameters (cont.)
10 Parameter Markers
11 Example
12 Late Binding of Parameters
13 Answers
14 A Query with **`SQLBindCol`**
15 A Query with **`SQLGetData`**
16 Describe Columns
17 Transactions
18 Summary

Call Level Interface/ODBC

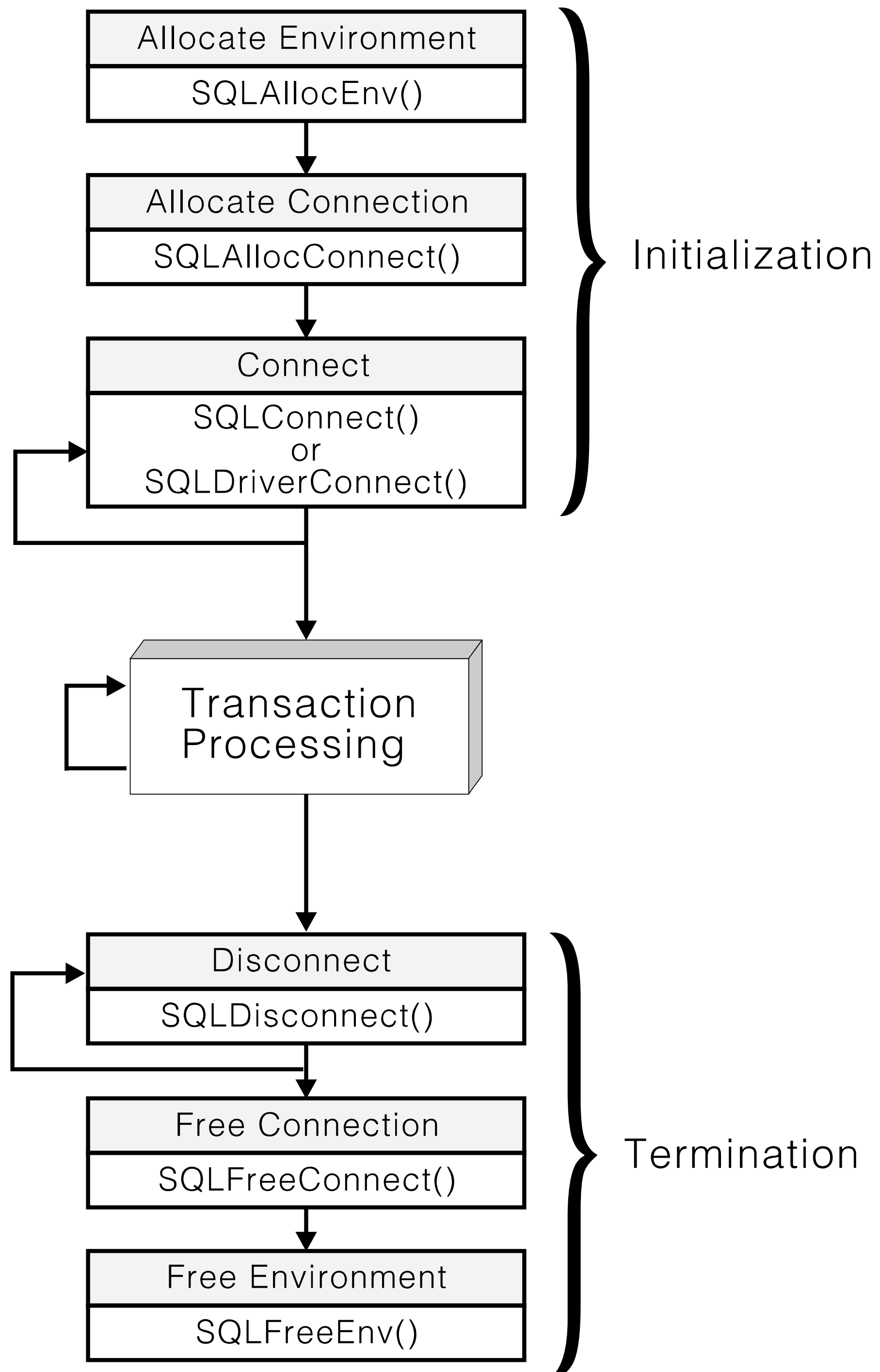
An interface built on a library calls:

- Applications are developed without access to the DB (and without additional tools: no precompilation)
- incorporates ODBC (M$) 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

ODBC Application Structure



Each object is referenced by a *handle* of type HENV, HDBC and HSTMT, respectively.

Every ODBC application must allocate exactly one environment.

Object Functions in ODBC:

For environments

- SQLAllocEnv (allocates object)
- SQLError (obtains diagnostics)
- SQLFreeEnv (frees object)

For connections

- SQLAllocConnect (allocates object)
- SQLConnect (connects to a database)
- SQLSetConnectOption (connection options)
- SQLTransact (used to COMMIT or ROLLBACK changes)
- SQLDisconnect (disconnect from a database)
- SQLError (obtains diagnostics)
- · · ·
- SQLFreeConnect (frees object)

Connect and Disconnect

```
#include <sqlcli1.h>

int main()
{
   SQLHENV   henv;
   SQLHDBC   hdbc;
   SQLRETURN rc;
   SQLCHAR   server[SQL_MAX_DSN_LENGTH + 1] = "SAMPLE";
   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);
       return (SQL_ERROR);
   } else
       printf("Connected to %s\n", server);

   /* DO SOMETHING HERE */

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

   return (SQL_SUCCESS);
}
```

Errors

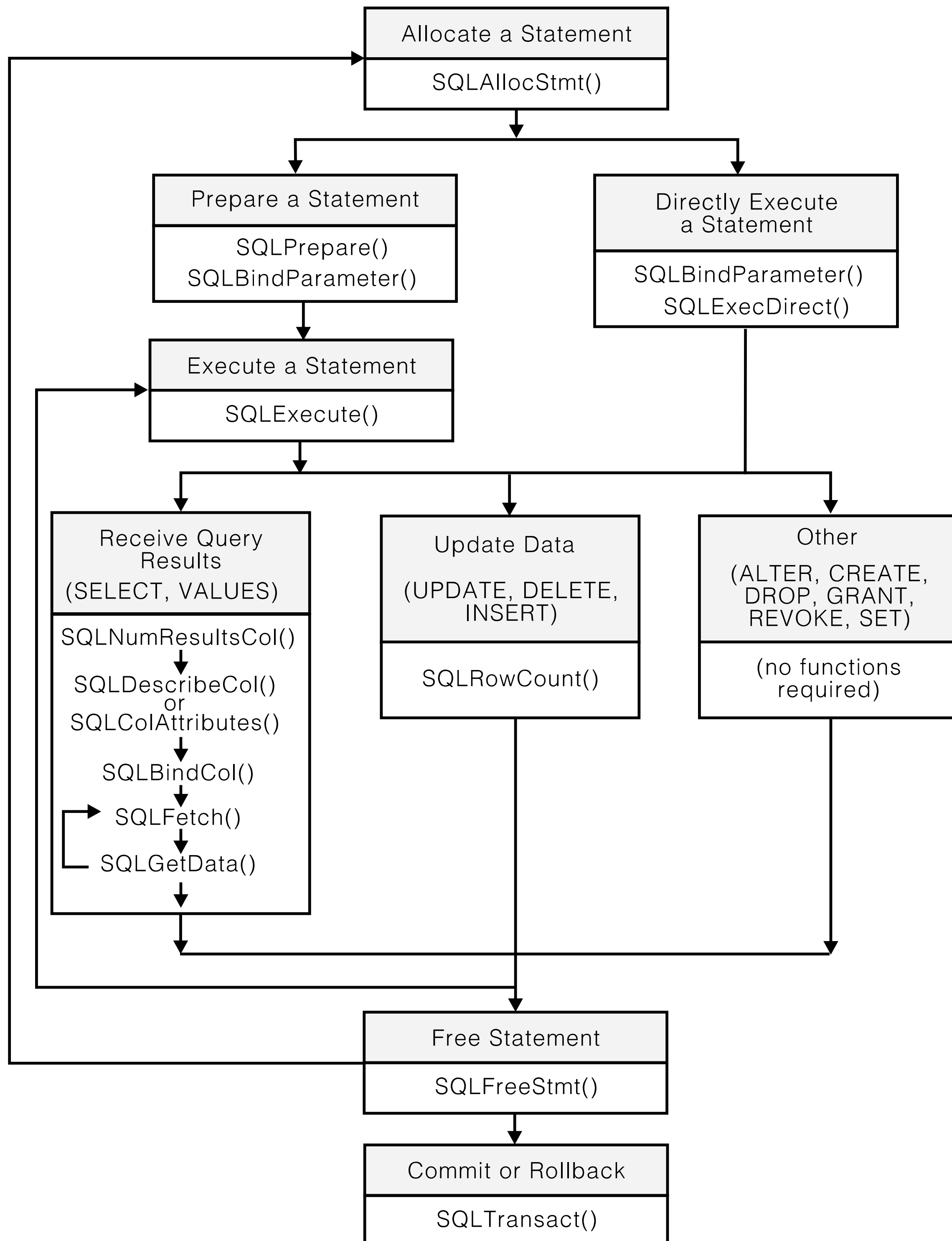
- `SQLxxx` functions return error codes
  - ⇒ similar to `libc` functions
  - ⇒ we should check them after every `SQLxxx` call
- the actual return codes:
  - ⇒ `SQL_SUCCESS`
  - ⇒ `SQL_ERROR`
- use the `SQLError` function to get sensible messages

SQL Statements

and what can we do with them: functions defined on statements:

- **`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)

Statement Life



Parameters

1. parameter markers

   '?' in the text of the query

   SQLNumParams

   SQLBindParameter

2. results of queries

   ⇒ specified by the number of resulting columns

   SQLNumResultsCol

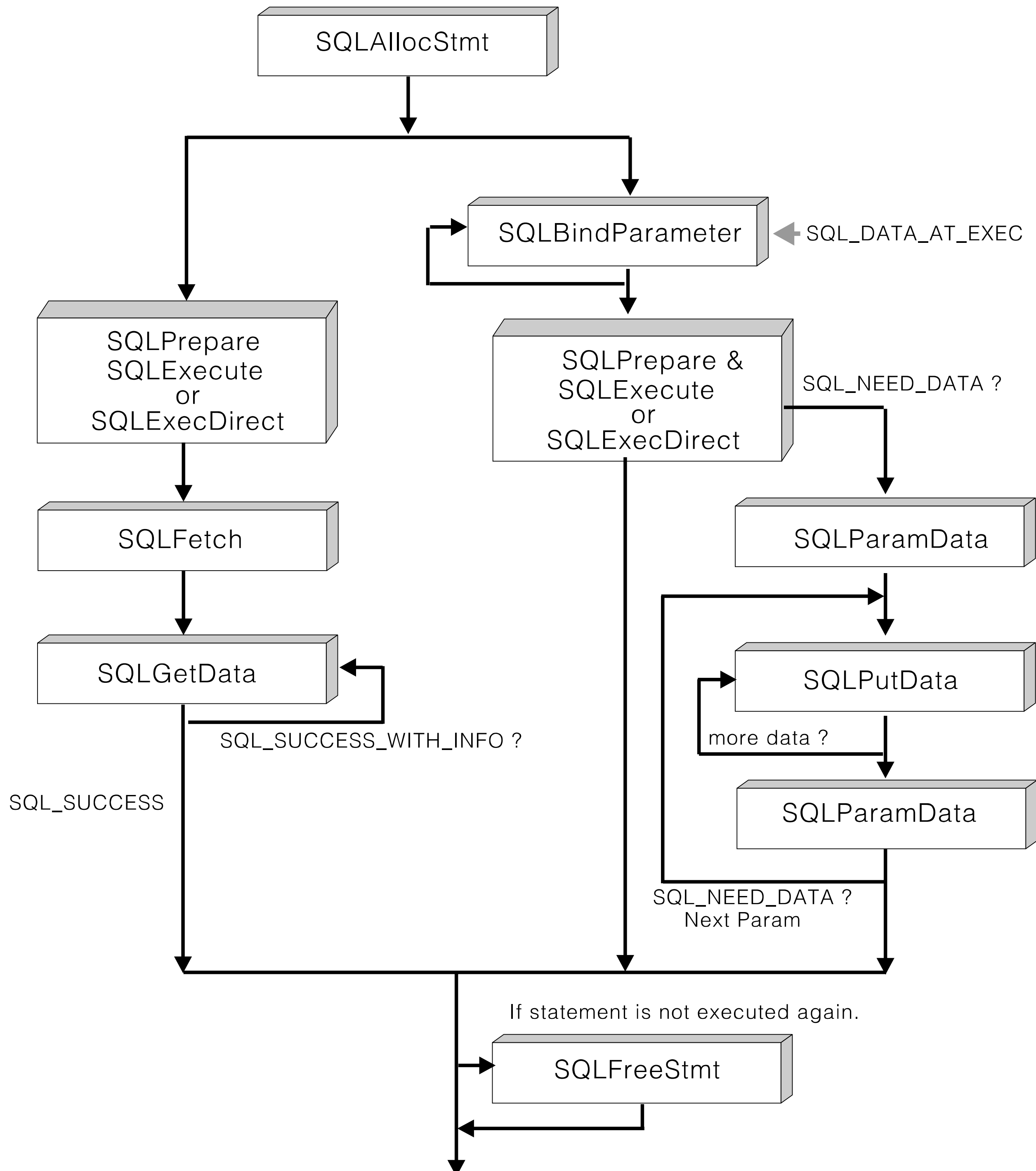
   SQLDescribeCol

   SQLBindCol or **SQLGetData**

3. number of affected tuples (updates):

   SQLRowCount

Parameters (cont.)



Parameter Markers

The parameters are **bound** using:

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

$\Rightarrow$ 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;

struct { SQLINTEGER ind;
         SQLCHAR    s[70];
       } url;

rc = SQLAllocStmt(hdbc, &hstmt);

rc = SQLPrepare(hstmt, stmt, SQL_NTS);

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

rc = SQLBindParameter(hstmt, 1,
              SQL_PARAM_INPUT, SQL_C_CHAR,
              SQL_CHAR, 0, 0, &url, 70, NULL);

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

Answers

Output values from a statement:

- number of affected: `SQLRowCount`
- answers to queries:
  1. bind variables before execution: `SQLBindCol`
  2. fetch values after execution: `SQLGetData`

NOTE: 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*

  $\Rightarrow$ no precompilation
  $\Rightarrow$ explicit binding of parameters
  user's has to make types match!

- An almost standard (ODBC, X/Open)

  $\Rightarrow$ independence on DBMS
  $\Rightarrow$ but: the standard has 100's of functions