Schedule

- Oct. 30 (T) Embedded SQL.
  - Read Section 8.1.
  - Assignment 5 due. Not accepted late.
  - Project Part 4 due on Sat Nov 3.
- Oct. 31 (W) 7–9pm: Review Session. Location TBA.
- Nov. 1 (TH) Midterm
  - Covers material through Oct. 25 (TH) lecture and readings (Chapters 1-3, 5-7).
- Nov. 6 (T) Transactions, Authorization.
  - Read Sections 8.6-8.7.
- Nov. 8 (TH) Object-Oriented Database Design.
  - Read Sections 4.1-4.4. Project Part 5 due.
Cursors

Declare by:

```
CURSOR <name> IS
    select-from-where statement
```

- Cursor gets each tuple from the relation produced by the select-from-where, in turn, using a `fetch statement` in a loop.
  - Fetch statement:
    ```
    FETCH <cursor name> INTO variable list;
    ```
  - Break the loop by a statement of the form:
    ```
    EXIT WHEN <cursor name> %NOTFOUND;
    ```
    - True when there are no more tuples to get.
- Open and close the cursor with `OPEN` and `CLOSE`. 
Example

A procedure that examines the menu for Joe’s Bar and raises by $1.00 all prices that are less than $3.00.

Sells(bar, beer, price)

- This simple price-change algorithm can be implemented by a single UPDATE statement, but more complicated price changes could not.
CREATE PROCEDURE joeGouge() AS
  theBeer Sells.beer%TYPE;
  thePrice Sells.price%TYPE;
  CURSOR c IS
    SELECT beer, price
    FROM Sells
    WHERE bar = 'Joe''s bar';
BEGIN
  OPEN c;
  LOOP
    FETCH c INTO theBeer, thePrice;
    EXIT WHEN c%NOTFOUND;
    IF thePrice < 3.00 THEN
      UPDATE Sells
      SET price = thePrice + 1.00
      WHERE bar = 'Joe''s Bar'
        AND beer = theBeer;
  END IF;
  END LOOP;
  CLOSE c;
END;
run
Row Types

Anything (e.g., cursors, table names) that has a tuple type can have its type captured with %ROWTYPE.

• We can create temporary variables that have tuple types and access their components with dot.

• Handy when we deal with tuples with many attributes.
Example

The same procedure with a tuple variable bp.

```
CREATE PROCEDURE joeGouge() AS
    CURSOR c IS
        SELECT beer, price
        FROM Sells
        WHERE bar = 'Joe''s bar';

    bp c%ROWTYPE;
BEGIN
    OPEN c;
    LOOP
        FETCH c INTO bp;
        EXIT WHEN c%NOTFOUND;
        IF bp.price < 3.00 THEN
            UPDATE Sells
            SET price = bp.price + 1.00
            WHERE bar = 'Joe''s Bar'
            AND beer = bp.beer;
        END IF;
    END LOOP;
    CLOSE c;
END;
```

run
Embedded SQL

Add to a conventional programming language (C in our examples) certain statements that represent SQL operations.

• Each embedded SQL statement introduced with EXEC SQL.

• Preprocessor converts C + SQL to pure C.
  • SQL statements become procedure calls.
Shared Variables

A special place for C declarations of variables that are accessible to both SQL and C.

• Bracketed by
  
  EXEC SQL BEGIN/END DECLARE SECTION;

• In Oracle Pro/C (not C++) the “brackets” are optional.

• In C, variables used normally; in SQL, they must be preceded by a colon.
Example

Find the price for a given beer at a given bar.

Sells(bar, beer, price)

EXEC SQL BEGIN DECLARE SECTION;
  char theBar[21], theBeer[21];
  float thePrice;
EXEC SQL END DECLARE SECTION;

  /* assign to theBar and theBeer */
  . . .

/* assign to theBar and theBeer */
  . . .

EXEC SQL SELECT price
  INTO :thePrice
FROM Sells
WHERE beer = :theBeer AND
  bar = :theBar;
  . . .
Cursors

Similar to PL/SQL cursors, with some syntactic differences.

Example

Print Joe’s menu.

```sql
Sells(bar, beer, price)
EXEC SQL BEGIN DECLARE SECTION;
    char theBeer[21];
    float thePrice;
EXEC SQL END DECLARE SECTION;
EXEC SQL DECLARE c CURSOR FOR
    SELECT beer, price
FROM Sells
    WHERE bar = 'Joe''s Bar';
EXEC SQL OPEN CURSOR c;
while(1) {
    EXEC SQL FETCH c
        INTO :theBeer, :thePrice;
    if(NOT FOUND) break;
/* format and print beer and price */
}
EXEC SQL CLOSE CURSOR c;
```
Oracle Vs. SQL Features

• SQL expects FROM in fetch-statement.

• SQL defines an array of characters SQLSTATE that is set every time the system is called.
  ◆ Errors are signaled there.
  ◆ A failure for a cursor to find any more tuples is signaled there.
  ◆ However, Oracle provides us with a header file sqlca.h that declares a communication area and defines macros to access it.
  ◆ In particular, NOT FOUND is a macro that says “the no-tuple-found signal was set.”
Dynamic SQL

Motivation:

• Embedded SQL is fine for fixed applications, e.g., a program that is used by a sales clerk to book an airline seat.

• It fails if you try to write a program like sqlplus, because you have compiled the code for sqlplus before you see the SQL statements typed in response to the SQL> prompt.

• Two special statements of embedded SQL:
  ◆ PREPARE turns a character string into an SQL query.
  ◆ EXECUTE executes that query.
Example: Sqlplus Sketch

EXEC SQL BEGIN DECLARE SECTION;
  char query[MAX_QUERY_LENGTH];
EXEC SQL END DECLARE SECTION;

/* issue SQL> prompt */

/* read user's text into array query */

EXEC SQL PREPARE q FROM :query;
EXEC SQL EXECUTE q;

/* go back to reissue prompt */

• Once prepared, a query can be executed many times.
  ◆ “Prepare” = optimize the query, e.g., find a way to execute it using few disk-page I/O’s.

• Alternatively, PREPARE and EXECUTE can be combined into:

  EXEC SQL EXECUTE IMMEDIATE :query;
Call-Level Interfaces

A more modern approach to the host-language/SQL connection is a call-level interface, in which the C (or other language) program creates SQL statements as character strings and passes them to functions that are part of a library.

- Similar to what really happens in embedded SQL implementations.
- Two major approaches: SQL/CLI (standard of ODBC = open database connectivity) and JDBC (Java database connectivity).
CLI

- In C, library calls let you create a *statement handle = struct in which you can place an SQL statement.*
  - See text. See also Monjian book for PostgreSQL.
- Use `SQLPrepare(myHandle, <statement>, ...)` to make `myHandle` represent the SQL statement in the second argument.
- Use `SQLExecute(myHandle)` to execute that statement.

Example

```sql
SQLPrepare(handle1, "SELECT ~beer, ~price
            FROM Sells
            WHERE bar = 'Joe''s Bar' ");
SQLExecute(handle1);
```
Fetching Data

To obtain the data returned by an executed query, we:

1. Bind variables to the component numbers of the returned query.
   - `SQLBindCol` applies to a handle, column number, and variable, plus other arguments (see text).

2. Fetch, using the handle of the query’s statement.
   - `SQLFetch` applies to a handle.

Example

```c
SQLBindCol(handle1, 1, SQL_CHAR, &theBar,...)
SQLBindCol(handle1, 2, SQL_REAL, &thePrice,...)
SQLExecute(handle1);
...
while(SQLFetch(handle1) != SQL_NO_DATA) {
    ...
}
JDBC

- Start with a *Connection* object, obtained from the DBMS (see text).
- Method *createStatement()* returns an object of class *Statement* (if there is no argument) or *PreparedStatement* if there is an SQL statement as argument.

**Example**

```java
Statement stat1 = myCon.createStatement();
PreparedStatement stat2 =
    myCon.createStatement(
        "SELECT beer, price " +
        "FROM Sells" +
        "WHERE bar = 'Joe''s Bar'"
    );
```

- *myCon* is a connection, *stat1* is an “empty” statement object, and *stat2* is a (prepared) statement object that has an SQL statement associated.
Executing Statements

• JDBC distinguishes queries (statements that return data) from *updates* (statements that only affect the database).

• Methods *executeQuery()* and *executeUpdate()* are used to execute these two kinds of SQL statements.
  ♦ They must have an argument if applied to a *Statement*, never if applied to a *PreparedStatement*.

• When a query is executed, it returns an object of class *ResultSet*.

**Example**

```java
stat1.executeUpdate("INSERT INTO Sells" + "VALUES('Brass Rail', 'Bud', 3.00)"
);
ResultSet Menu = stat2.executeQuery();
```
Getting the Tuples of a *ResultSet*

- Method *Next()* applies to a *ResultSet* and moves a “cursor” to the next tuple in that set.
  - Apply *Next()* once to get to the first tuple.
  - *Next()* returns *FALSE* if there are no more tuples.
- While a given tuple is the current of the cursor, you can get its *ith* component by applying to a *ResultSet* a method of the form *get X(i)*, where *X* is the name for the type of that component.

**Example**

```java
while(Menu.Next()) {
    theBeer = Menu.getString(1);
    thePrice = Menu.getFloat(2);
    ...
}
```