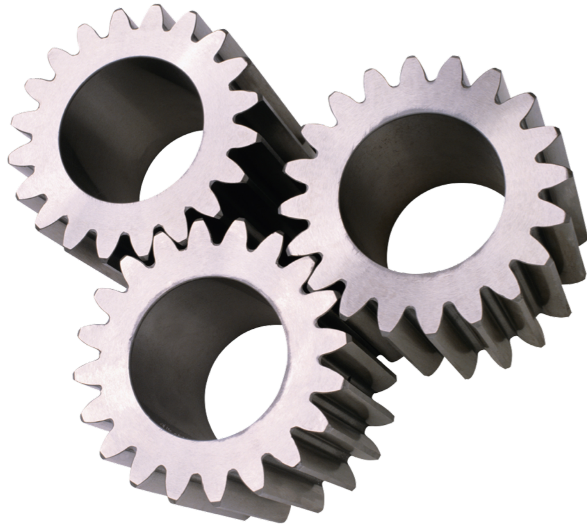
COSC344
Database Theory and Applications



Lecture 13: C and SQL

Overview

- Last Lecture
  - Java SQL
- This Lecture
  - C & SQL
  - Source: Lecture notes,
    Textbook: Chapter 10
    Program examples
- Next Lecture
  - PHP & SQL

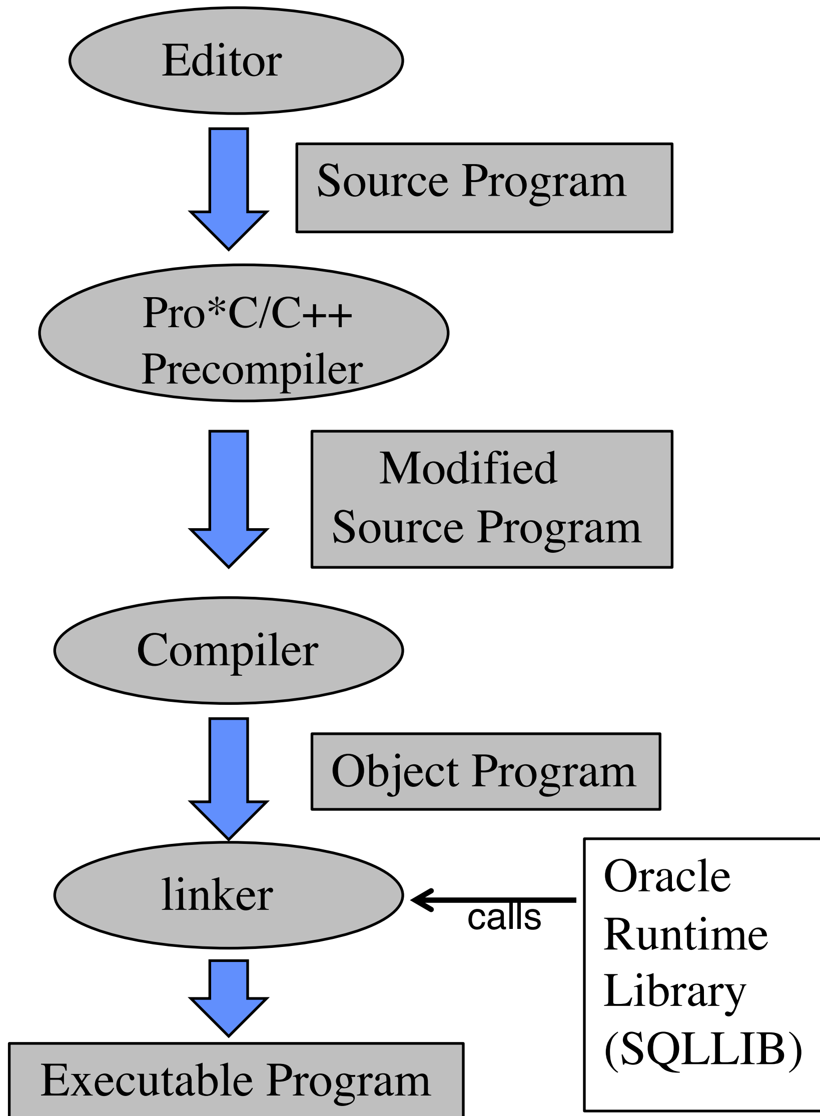
Embedded SQL

- Embed database statement in a host language (e.g. C/C++)
- Identified by a special prefix (`EXEC SQL`)
- Terminated by a semicolon (;)

```
EXEC SQL
  SELECT fname, lname, dno
  FROM employee;
```

- A precompiler or preprocessor identifies the database statements and extract them for processing in DBMS
- Pro* C/C++ precompiler
  - C/C++ and SQL programming to manipulate data in Oracle

Embedded SQL Program Development



- Editor
  - Host language program with embedded SQL statements
  - <file name>.pc
- Precompiler
  - Replaces SQL statements with Oracle provided library calls
  - <file name>.c
- Compiler
  - Generates an object file
  - <filename>.o
- Linker
  - Resolves references to Oracle runtime libraries, system libraries
  - Executable program

Embedded SQL Statements

- Executable Statements
  - Result in calls to the runtime library SQLLIB
  - Can be placed wherever executable **host language statements** can be placed
  - Connect to Oracle, control access to Oracle data, process transactions, etc.
  - Examples - SELECT, INSERT, COMMIT, DELETE
- Declaratives
  - Do not result in calls to SQLLIB, not operate on Oracle Data
  - Can be placed wherever **host language variable declarations** can be placed
  - Declare objects, communication areas, and SQL variables
  - Example – DECLARE, INCLUDE , VAR, WHENEVER

Host Variables

- Declared in host program and shared with Oracle
- Used to communicate between host program and database
- Can be used anywhere an SQL expression can be used
- Must be prefixed with a colon (:)
- Can have the same name as database columns
- Must have data types compatible with the data types in the database
- Can use a *struct* to contain a number of host variables
  - Oracle uses each component of the struct as a host variable

Declaring Host Variables

- Declared within an embedded SQL `DECLARE` section

```
EXEC SQL BEGIN DECLARE SECTION;

    varchar  h_fname[16];
    varchar  h_lname[16];
    int      dno;
    char     h_ird[10];

EXEC SQL END DECLARE SECTION;
```

- NOTE - Oracle's embedded SQL does not require the `DECLARE` section. It only requires declaring host variables as in normal C. BUT the `DECLARE` section is part of the SQL standard. Some precompilers will issue errors if not included.

INTO Clause

- Specifies the host language target variables
- Retrieved values are placed into the specified host variables
- 1:1 correspondence between host variables and attributes in the SELECT clause
  - The ith retrieved value in the SELECT clause is placed in the ith host variable in the INTO clause

```
EXEC SQL
  SELECT fname, lname, dno
    INTO :h_fname, :h_lname, :h_dno
    FROM employee
    WHERE ird = :h_ird;
```

- Can be used only when the query result is a single record

VARCHAR Data Type

- A special type provided by ORACLE as a predeclared struct
- The precompiler changes this declaration

```
VARCHAR username[20];
```

into the following struct

```
struct {
  unsigned short len;
  unsigned char  arr[20];
}  username;
```

- Two components

```
username.len
username.arr
```

- Do not forget to allow for '\0’

Why VARCHARS?

- You can explicitly reference the length of a returned string after a SELECT or FETCH
- ORACLE does not explicitly terminate strings with the null terminator
- ORACLE puts the length of the character string into the length member.
- You can then use this length to add the null terminator

```
username.arr[username.len] = '\0';
```

- Or you can use the length in a strncpy or printf statement

```
printf("Username is %.*s\n", username.len,
                             username.arr);
```

Delimiters

- C
  - Use single quotes to delimit single characters

```
ch=getchar();
Switch (ch)
{ case 'U': update(); break;
  … }
```

  - Use double quotes to delimit character strings

```
printf("Good Morning");
```

- SQL
  - Use single quotes to delimit character strings

```
EXEC SQL SELECT salary WHERE fname='John';
```

  - Use double quotes to delimit identifiers containing special or case-sensitive characters

```
EXEC SQL CREATE TABLE "Emp2" (…)
```

Database Connection

- Standard CONNECT
  - `EXEC SQL CONNECT :username IDENTIFIED BY :password;`

    username and *password* are **char** or VARCHAR host variables
  - `EXEC SQL CONNECT :usr_pwd;`

    usr_pwd contains username and password separated by a slash character (/).

- Change Password on CONNECT
  - `EXEC SQL CONNECT .. ALTER AUTHORIZATION :newpswd;`
    - Change the account password to `newpswd`
    - Connect to database using `user/newpswd`

Close Connection

- Exit gracefully if the last SQL statement it executes is either

```
EXEC SQL COMMIT WORK RELEASE;
```

or

```
EXEC SQL ROLLBACK WORK RELEASE;
```

Example 1 - getemp

- Small sample program that demonstrates
  - DECLARE section
  - Connecting to the database
  - A very basic query
  - Disconnecting from the database in a tidy way
- Task

  Asks for a IRD, and retrieves the first and last name and department number from the EMPLOYEE table.
- Source code: /coursework/344/pickup/oracle-C/getemp.pc

Errors and Recovery

- Status Variables: SQLCODE and SQLSTATE
  - DBMS returns a value in SQLCODE after each database command is executed.
  - Should be tested after every SQL statement and take appropriate action if the value is not what was expected
- The SQL Communications Area (SQLCA)
  - Contains components that are filled in at runtime after SQL statement is processed by Oracle

SQLCODE

- A long integer variable declared either inside or outside of the Declare section
- In Oracle, it is a field in SQLCA – *sqlca.sqlcode*

```
=0   statement executed without error or exception

>0   statement executed but detected an exception.
     Examples are: WHERE clause condition not met
     or no rows returned

<0   statement NOT executed because of an application,
     database, system, or network error
```

There is a complete list in Oracle8*i Error Messages*

SQLSTATE

- A **5-characters null-terminated string**
- Must be declared inside the Declare section

```
char SQLSTATE[6]; /* Upper case is required. */
```

- Stores both error and warning codes
- The reporting mechanism uses a standardized coding scheme.

```
00000   success completion

02000   no data

08000   connection exception

08003   connection does not exist

22012   division by zero
```

……

Status Variables Example

```
EXEC SQL BEGIN DECLARE SECTION;
  varchar fname[16], lname[16];
  Char ird[10];
  int dno, dnumber;
  int SQLLCODE; char SQLSTATE[6];
EXEC SQL END DECLARE SECTION;
```

```
EXEC SQL SELECT Fname,Lname
         INTO :fname, :lname
         FROM EMPLOYEE WHERE ird = :ird;
If (SQLCODE == 0) /*or (strncmp(SQLSTATE,"00000",6)) */
     printf(fname, lname, ird)
else
     printf(“Ird does not exist!”)
```

SQLCA

- SQLCA (SQL Communications Area) is an ORACLE data structure
- It holds status information provided by ORACLE after each statement is executed
  - ORACLE error codes
  - warning flags
  - count of rows processed
  - Diagnostics
- SQLCA is defined in the header file *sqlca.h*

SQLCA struct

```
struct sqlca
{
  char    sqlaid[8];    → Identify the SQL Communication Area
  long    sqlabc;       → The length(in bytes) of the SQLCA structure
  long    sqlcode;      → The status code of the most recently executed SQL statement
  struct
    {
      unsigned short  sqlerrml;    → Length of the message text in sqlrrmc
      char            sqlerrmc[70];→ Text corresponding to the error code
    } sqlerrm;
  char    sqlerrp[8];   → Reserved for future use
  long    sqlerrd[6];   → Sqlerrd[2] holds the number of rows processed
  char    sqlwarn[8];   → Warning flags
  char    sqlext[8];    → Reserved for future use
}
```

sqlglm() Function

- *sqlglm()* can be used to get the message in *sqlerrmc*

```
void sqlglm (char   *buffer, size_t *buffer_size,
                     size_t *return_length)

buffer - text buffer into which you want Oracle to
         store the error message. Oracle blank-pads
         to the end of the buffer.

buffer_size - specifies the maximum size of the
              buffer in bytes

return_length - a variable into which Oracle stores
                the actual length of the error message
```

Example 2 - upemp

- Asks for an employee's IRD, retrieves and displays the first and last names and current salary, asks for a new salary, and updates the database.

- Checks for errors

- Source code: /coursework/344/oracle-C/upemp.pc

Host Variable Indicator

- Indicator variable is useful for
  - In VALUES or SET clause to assign NULLs to input host variable
  - In INTO clause to detect NULLs
- Syntax:

```
    :host_variable INDICATOR :indicator_variable
OR
    :host_variable:indicator_variable
```

- Each time the host variable is used in a SQL statement, a result code is stored in its associated indicator variable.
  - On Input

-1	Assign a NULL to the column, ignoring the value of host variable
>=0	Assign the value of the host variable to the column

  - On Output

0	The operation was successful
-1	A NULL was returned, inserted, or updated.

Host Variable Indicator (cont.)

- Inserting NULLs

```
short ind_comm = -1;
EXEC SQL INSERT INTO employee (empno,comm)
     VALUES (:emp_number, :commission:ind_comm);
```

- Handling Returned NULLs

```
EXEC SQL SELECT ename, sal, comm      INTO
:emp_name, :salary, :commission:ind_comm From
employee WHERE empno = :emp_number;
if (ind_comm == -1)
   pay = salary;
Else pay = salary + commission;
```

- Testing for NULLs

```
EXEC SQL SELECT ename, sal
INTO :emp_name, :salary
FROM employee
WHERE :commission INDICATOR :ind_comm IS NULL:
```

A New Problem

- The previous examples operated on a single row of a table
- Many queries will return multiple rows to be processed
- Programming languages
  - Operate on a tuple at a time
- Database Management Systems
  - Return a table at a time
  - So far, the returned tables had only one row
- Retrieving multiple tuples using cursors

Cursors

- When a query returns multiple rows, defining a cursor allows us to
  - process beyond the first row returned
  - keep track of which row is currently being processed
- Cursors are defined and manipulated using
  - DECLARE
  - OPEN
  - FETCH
  - CLOSE

Declaring Cursors

- Syntax

```
EXEC SQL
   DECLARE <cursor name> CURSOR FOR
   <select-expression>;
```

- Cursor name - similar to a pointer variable
- There is no INTO clause
- Example

```
EXEC SQL
   DECLARE emp_cursor CURSOR FOR
     SELECT fname, lname, dno
     FROM employee
     WHERE fname LIKE 'J%';
```

Opening a Cursor

- Syntax

```
EXEC SQL OPEN <cursor name>;
```

- Example

```
EXEC SQL OPEN emp_cursor;
```

- Opens a cursor (which must be closed)
- Gets the query result from the database
- The rows returned become the cursor's current active set
- Sets the cursor to position before the first row. This becomes the current row.
- NOTE - You must use the same cursor name if you want data from that cursor.

Fetching A Row

- Syntax

```
EXEC SQL
  FETCH <cursor name>
  INTO <host variables>;
```

- Example

```
EXEC SQL
  FETCH emp_cursor
  INTO :h_fname, :h_lname, :h_dno;
```

- Moves the cursor to the next row in the current active set
- Assigns values to the host variables

Closing the Cursor

- Syntax

```
EXEC SQL CLOSE <cursor name>;
```

- Example

```
EXEC SQL CLOSE emp_cursor;
```

- Closes the cursor (which must be open)
- There is no longer an active set
- Reopening the same cursor will reset it to point to the beginning of the returned table

Example 3

- Retrieve the first and last name and department number from the EMPLOYEE table.
  - Source code: /coursework/344/oracle-C/empname.pc
- Retrieve the project name and project number for a given project location
  - A user-friendly interface: a menu
  - Source code: /coursework/344/oracle-C/query_c_locations.pc