CS 321: Database Systems

Rahul Simha

Department of Computer Science
College of William & Mary
Williamsburg, VA

Chapter 1

Oracle

CS 321: Database Systems

1.1 Oracle: An Introduction

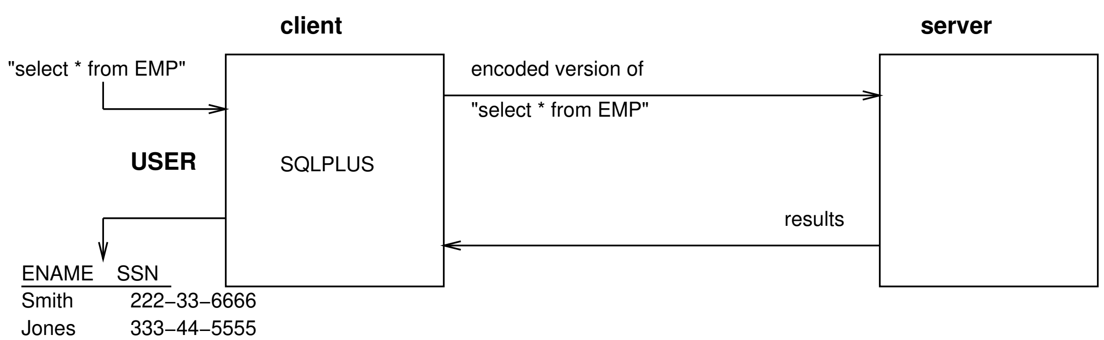
- What is Oracle?
  ⇒ Oracle is a commercial database product.

  Oracle consists of:

  1. A DBMS server:
     - Parses and executes SQL.
     - Executes "stored" PL/SQL code.
     - Handles query processing and optimization.
     - Handles disk I/O, file system and memory management (paging).
     - Provides concurrency, recovery and security.
  2. Database tools:
     - DBA tools: svrmgrl (line mode) and svrmgrm (window mode).
       * System startup and monitoring.
       * Add/delete users, set priveleges for users.
       * Control space usage.
     - SQL interpreter: sqlplus
       * Prompt user for SQL input and connect to server.
       * Pass SQL queries to server and retrieve results.
       * Other features: formatting, transaction processing, Oracle extensions to SQL.
       * PL/SQL compiler.
     - A dbase programming language: PL/SQL
       * Full (almost) programming language.
       * Syntax and keywords based on Ada.
       * Standard libraries for process control, transaction processing, I/O.

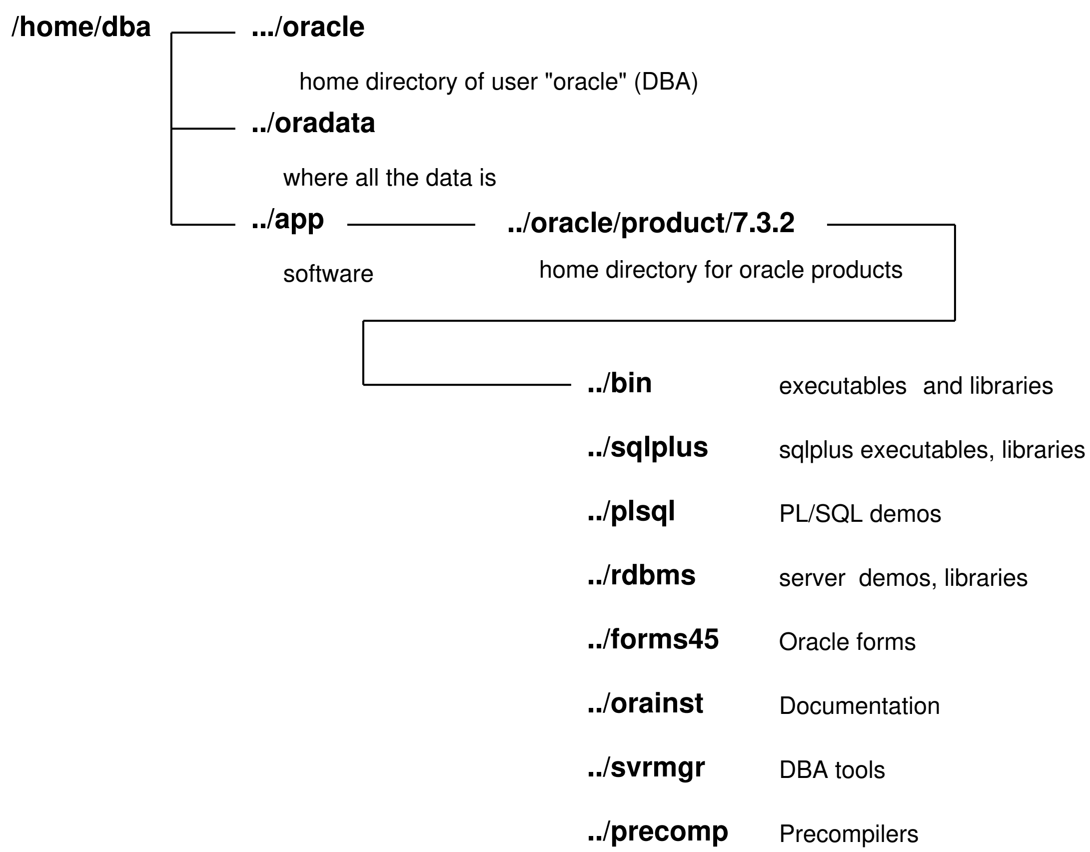
- A window-based programmable user-interface: OracleForms
    * Can set up window-based environment for end-users.
    * Customizable and programmable.
- Other tools:
    * A report-generating and formatting tool: OracleReport.
    * A tool for designing web-based interfaces: Webserver.
    * A tool for handling spatial data.
    * Tools for distributed data, parallel execution of server, graphics and multimedia.

- Oracle is designed to work in client-server mode:
  ⇒ the tools operate as server clients.



- Oracle supports all of SQL/92 (the most recent ANSI SQL standard).

- Oracle on Unix:
    - Oracle is usually under a directory called `/home/dba/`.
    - The directory structure in the standard installation looks like:



- Oracle files and tablespaces:
  - Oracle keeps data in a bunch of tablespaces:
    $\Rightarrow$ a tablespace is a group of related data
    For example, all students have their data in tablespace `STUDENT97`.
  - Each tablespace consists of a number of (unix) files.
  - A relation is placed in a tablespace and may spread across several files.
  - The files are in `/home/dba/oradata/`.
  - Each user is required to have a:
    * Default tablespace: where the user's data is stored.
    * Temp tablespace: for intermediate computations (e.g., sorting).
  - Standard tablespaces: SYSTEM, TEMP, TOOLS, USER.
- Oracle processes: an Oracle server installation will have several processes running simultaneously:

- DBWR: Database writer – for disk I/O.
- LGWR: Log writer – for writing recovery information.
- SMON, PMON: System monitor and process monitor – to handle connections, client-server communication.
- Server processes.

- Oracle handles its own memory management: a huge piece of main memory (System Global Area) is kept for:
  - Data and catalogs.
  - Current and "stored" SQL and PL/SQL.
  - Memory for indexing, processing, recovery.

1.2 Oracle Datatypes

- Oracle has several datatypes, among which are:
  - CHAR(n) – Fixed-length character string.
    * n is the length.
    * Default length is 1.
    * Length must be between 1 and 255 bytes.
    * Fixed amount of space is allocated.
  - VARCHAR2(n) – variable-length character string.
    * n is the maximum length (must be specified). $1 \leq n \leq 2000$.
    * Space usage depends on length of string.
    * Use VARCHAR2 where possible.
  - VARCHAR(n) – currently same as VARCHAR2.
    * VARCHAR2 and VARCHAR correspond to the ANSI definition of VARCHAR.
    * VARCHAR may change with the next change in ANSI standards.
  - NUMBER(m, n) – fixed and floating point numbers.
    * m is the total number of digits.
    * n is the number of digits after the decimal point.
    * NUMBER($*$,n) specifies decimal accuracy but unlimited size.
    * Numbers are in the range 10^{-130} to 10^{125}.
  - INTEGER – integers.
  - DATE – to store a dates (year,month,day,time).
    * Date range: Jan 1, 4712 BC to Dec 31, 4712 AD.
    * A library of date-manipulation functions is provided.

- LONG – to store large amounts of text data.
  * Up to 2GB of text data.
  * Only *one* LONG column is permitted per relation.
  * LONG attributes cannot be used in the **where** subclause and in certain other places.
- RAW(n) – to store non-text data.
  * n is size of field: $1 \leq n \leq 255$.
  * RAW data is not converted into any format.
- LONG RAW – to store images, graphics etc.

• For comparison, here are some standard datatypes in SQL/92:

> CHAR, VARCHAR, NCHAR, NATIONAL CHAR, BIT, NUMERIC, DECIMAL, DEC, INTEGER, INT, SMALLINT, FLOAT, REAL, DOUBLE PRECISION, DATE, TIME, TIMESTAMP.

1.3 Oracle and SQL

- Oracle provides the tool `sqlplus` to execute SQL statements:
  - You can type SQL statements at the Sqlplus the command line directly:

    ```
    % sqlplus <username>/<password>
    SQL> select * from EMP;
    ```

  - Alternatively, you can place SQL statements in a file and execute them by specifying the file name, e.g.,
    - Suppose the file `test1.sql` contains the the string "`select * from EMP`".
    - To execute the SQL:

      ```
      SQL> @test1.sql
      ```

- Sqlplus has number of Oracle-specific commands and features:
  - Use `help` to obtain a complete list of commands.
  - The command `describe` tells you the structure of a table, e.g., to find out the definition of relation EMP:

    ```
    SQL> desc EMP
    ```

  - Output can be directed to a file using `spool`, e.g.,

    ```
    SQL> spool test1.output
    SQL> @test1.sql
    SQL> spool off
    ```

  - Sqlplus supports the definition and use of variables, e.g.,

```
define name_var = 'Smith'
select E.SSN
from EMP E
where E.NAME = '&name_var';
```

If a variable name is used without definition, e.g.,

```
select E.SSN
from EMP E
where E.NAME = '&name_var';
```

then Sqlplus prompts the user for the value.

– Sqlplus has features for rudimentary screen I/O, e.g.,

```
clear screen
accept name_var prompt Enter an employee name:

select E.SSN
from EMP E
where E.NAME = '&name_var';
```

- Oracle SQL differs from standard SQL in some ways, e.g., Oracle uses **minus** instead of **except** for set difference.
- Some additional useful facts:
  - Use '`;`' (semi-colon) to see the last command executed.
    ⇒ command is in server's SQL buffer
    ⇒ already parsed and ready for execution.
  - Use '`\`' (backslash) to execute last command.
  - Use '`!`' (exclamation) before Unix command to execute a Unix command.
  - To see a list of tables that you have created:
    `select table_name from user_tables;`

– System tables that may be useful to you: `all_tables`, `all_users`, `dba_free_space`, `dba_tablepsaces`, `dba_users`.

1.4 PL/SQL: Introduction

- Consider the following relation:

TRAVEL	FLT_NO	DEST	NEAREST	AIRLINE
	139	Montreal	New York	Air Canada
	452	Brussels	London	Sabena
	339	London	New York	British Airways
	116	Macau	Hong Kong	Macau Air
	322	Hong Kong	Tokyo	Japan Air Lines
	43	Tokyo	San Francisco	USAir
	31	San Francisco	New York	McValue

  – Suppose we are given a destination and we want to produce a list of flights starting from New York to that destination.
  – We can't do this in SQL.
    ⇒ cannot compute transitive closure in SQL
    ⇒ alternative programming language needed.

- Most DBMS's extend SQL with their own programming language.
- In Oracle: PL/SQL (Procedural Language extensions to SQL):
  – PL/SQL has (almost) the full power of many programming languages (like C).
  – PL/SQL's syntax and keywords are modeled on Ada.
  – PL/SQL has dbase-specific constructs such as *cursors* and *anchored declarations*.
  – PL/SQL has pre-defined libraries for I/O, process control and inter-process communication.
- "Hello World" in PL/SQL:
  – There is no separate tool for PL/SQL.

- PL/SQL can be entered at the command line of Sqlplus or run as a file.
- Suppose the file `test1.pl` contains:

```
begin
  dbms_output.enable;
  dbms_output.put_line('Hello World');
end;
/
```

- Then, the Sqlplus commands

```
SQL> set serveroutput on;
SQL> @test1.pl
 Hello World
```

  cause the file to be executed.

• In general a PL/SQL program will have the following structure:

```
declare
  /* variables, if any, are declared here */
begin
  /* code */
exception
  /* exception handling code -- optional */
end; /* Note the semi-colon */
/    /* Note the forward-slash */
```

• Datatypes in PL/SQL: all of Oracle's SQL datatypes are available in PL/SQL, and with the same names, e.g.,

char, varchar2, number, date, boolean, long,
raw

Example: the following are declarations in PL/SQL:

```
declare
  i integer;
  x number(10,3);
  ename varchar2(50);
begin
  null;
end;
```

- Anchored declarations:
  - Suppose I have the following declaration:

    ```
    emp_name varchar2(50);
    ```

  - Later, we want to make comparisons with a variable called `mgr_name`.
  - One option:

    ```
    emp_name varchar2(50);
    mgr_name varchar2(50);
    ```

  - Instead, it's better to let the type of `mgr_name` depend on `emp_name` using an *anchored declaration*:

    ```
    emp_name varchar2(50);
    mgr_name emp_name%TYPE;
    ```

  - Anchored declarations are even more useful in creating variables to hold tuples.
  - For example, suppose we have created (in SQL) a table called `Travel`:

create table Travel (	Flt_no	**integer**
	Dest	**varchar(50)**
	Nearest	**varchar(50)**
	Airline	**varchar(50)**);

To declare variables of the same types as above in PL/SQL:

```
fltnum Travel.Flt\_no%TYPE;
which_airline Travel.Airline%TYPE;
```

– It is even possible to define a variable to hold a complete tuple:

```
declare
  travel_tuple Travel%ROWTYPE;
begin
  ...
  dbms_output.put_line (travel_tuple.Airline);
  ...
end;
```

1.5 PL/SQL: Control Statements

- Like any standard high-level programming language, PL/SQL has conditional constucts:

```
if <cond> then
  /* statements */
end if;  /* Note the space between end and if */

if <cond> then
  /* blah-blah */
else
  /* rah-rah */
end if;

if <cond> then
  /* s1 */
elsif <cond>
  /* s2 */
elsif <cond>
  /* s3 */
else
  /* s4 */
end if;
```

- Loops:
  - The simplest loop in PL/SQL is of this form:

```
loop
  ...
```

```
  if <cond> then exit;
  ...
end loop;
```

– To simulate a C-while loop, for example

```
i = 1;
while (i <= n) {
  ...
}
```

use the following PL/SQL loop:

```
i := 1;
loop
  if (i > n) then exit;
  ...
end loop;
```

– Similarly, for example, the following C-forloop

```
for (i=1; i<=n; i++) {
  ...
}
```

can be simulated in PL/SQL as:

```
i := 1;
loop
  if (i > n) then
    exit;
  else
    i := 1 + 1;
  end if;
  ...
```

```
end loop;
```

- Note: several variants of loops exist in PL/SQL (see the manual).

1.6 PL/SQL: Cursors

- SQL by itself provides no for-loop like mechanism to iterate through the tuples of a relation.
- Yet, it is very useful to be able to retrieve tuples one-by-one for processing.
- Most DBMS vendors provide such mechanisms – often called cursors.
- Oracle provides cursors in PL/SQL as well as in the C interfaces.
- What exactly is a cursor?
  - To understand cursors, first consider a list of items in C++, e.g.,

```
struct list_item {
  ...
} item_type;

class list {
  ...
  // Access functions
  void start ();          // Initialize walk through list
  item item_next ();      // Return current item
  boolean list_empty (); // Is the list empty?
  boolean at_end ();      // Are we at the end?
};
```

  - This `list` class would be used as:

```
list A;
item_type item;
...
```

```
if (! A.list_empty () ) {
  A.start ();
  while (! A.at_end () ) {
    item = A.next ();
    ...
  }
}
```

- Note that the class `list` is a *generalization* of a simple for-loop variable.
- A cursor is similar:
  - Cursors provide ways to determine if a relation is empty, to start a scan and determine if the scan has ended.
  - A cursor is an "object" that scans a relation.
  - Several cursors can be defined for a single relation.
  - Each cursor applies to only one relation.
- In PL/SQL, you can:
  - Declare a cursor.
  - Open a cursor.
  - Use a cursor repeatedly.
  - Close a cursor.

- Example: consider the relation

  travel (Flt_no, Dest, Nearest, Airline).

  Let's write a PL/SQL program to "curse" through the relation:

```
DECLARE
  CURSOR travel_cursor IS   /* Declaration of cursor */
    SELECT *
    FROM Travel;
```

```
  travel_tuple Travel%ROWTYPE;
BEGIN

  DBMS_OUTPUT.ENABLE;

  OPEN travel_cursor;

  /* Fetch first tuple from travel relation */
  FETCH travel_cursor INTO travel_tuple;

  IF travel_cursor%FOUND THEN
    DBMS_OUTPUT.PUT ('First FLT\_NO found is: ');
    DBMS_OUTPUT.PUT_LINE (travel_tuple.Flt\_no);
  ELSE
    DBMS_OUTPUT.PUT_LINE ('Empty relation');
  END IF;

  CLOSE travel_cursor;

END;
```

Note:

- If the above code is in a file called `test2.pl`, it can be run from Sqlplus:

  ```
  SQL> set serveroutput on;
  SQL> @test2.pl
  ```

- The program doesn't do much: it only fetches one tuple.
- PL/SQL keywords are capitalized for reading convenience.

- Let's now write a more interesting program: to print out all the tuples in Travel.

```
DECLARE
  CURSOR travel_cursor IS   /* Declaration of cursor */
    SELECT *
    FROM Travel;

  travel_tuple Travel%ROWTYPE;

  i INTEGER;

BEGIN

  DBMS_OUTPUT.ENABLE;

  IF travel_cursor%ISOPEN THEN
    DBMS_OUTPUT.PUT_LINE ('Error: cursor already open');
  ELSE
    OPEN travel_cursor;
  END IF;

  /* Now curse through Travel */
  i := 1;
  LOOP
    FETCH travel_cursor INTO travel_tuple;
    IF travel_cursor%NOTFOUND THEN
      EXIT;
    END IF;
    DBMS_OUTPUT.PUT ('Tuple #');
    DBMS_OUTPUT.PUT (i);
    DBMS_OUTPUT.PUT (': ');
    DBMS_OUTPUT.PUT (travel_tuple.Flt\_no);
    DBMS_OUTPUT.PUT (travel_tuple.Dest);
    DBMS_OUTPUT.PUT (travel_tuple.Nearest);
    DBMS_OUTPUT.PUT_LINE (travel_tuple.Airline);
    i := i + 1
```

```
  END LOOP;

CLOSE travel_cursor;

END;
```

- Thus, the key steps in using a cursor are:
  1. Declaring a cursor via an SQL **select**, e.g.,

     ```
     CURSOR travel_cursor IS   /* Declaration of cursor */
       SELECT *
       FROM Travel;
     ```

  2. Opening a cursor, e.g.,

     ```
     IF travel_cursor%ISOPEN THEN
       DBMS_OUTPUT.PUT_LINE ('Error: cursor already open');
     ELSE
       OPEN travel_cursor;
     END IF;
     ```

     Note: it's always wise to check whether another cursor has been opened on the same relation.
  3. Using FETCH to obtain successive tuples.
  4. CLOSE-ing a cursor.
- Note that the **select** in a cursor declaration can be quite powerful, e.g.,

```
CURSOR travel_cursor IS   /* Declaration of cursor */
  SELECT *
  FROM Travel
  WHERE Travel.Airline = 'McValue'
  ORDER BY Travel.Dest;
```

This cursor will only retrieve those tuples satisfying the **where** condition.

Similarly, a cursor can be limited to a few attributes (via a projection):

```
DECLARE
  CURSOR travel_cursor IS   /* Declaration of cursor */
    SELECT Dest, Nearest
    FROM Travel
    WHERE Travel.Airline = 'McValue';

  travel_tuple travel_cursor%ROWTYPE
  /* travel_tuple only has 2 fields */
```

- The following special attributes are automatically available with any cursor, e.g., for `travel_cursor`:

travel_cursor%FOUND	Check if a tuple was retrieved
travel_cursor%NOTFOUND	See if nothing was retrieved
travel_cursor%ISOPEN	Is the cursor already open?
travel_cursor%ROWCOUNT	How many tuples are returned?

The last attribute counts the number of tuples that matched the defining **select** in the cursor definition.

1.7 PL/SQL: Procedures and Functions

- PL/SQL supports the definition and use of procedures and functions.
- Example:

```
PROCEDURE print_tuple_num (i IN INTEGER) IS
  /* Local variables are declared here */
BEGIN
     DBMS_OUTPUT.PUT ('Tuple #');
     DBMS_OUTPUT.PUT (i);
     DBMS_OUTPUT.PUT (': ');
END;

DECLARE
  ...
BEGIN
  ...
  print_tuple_num (i);
  ...
END;
```

Note:

- The general definition of a procedure has this structure:

```
PROCEDURE <name> ( <parameters> )
  <local variables>
BEGIN
  <body>
EXCEPTION
  <exception code>
```

```
END;
```

- Each parameter, apart from having a data type, also has an input type: either IN, OUT or IN OUT.
- Example:

```
PROCEDURE proc1 (x IN INTEGER, y OUT INTEGER, z IN OUT INTEGER)
```

- Functions can be defined which return a value. The general form is

```
FUNCTION <name> ( <parameters> )
  RETURN <return type>
IS
  <local variables>
BEGIN
  <body>
EXCEPTION
  <exception code>
END;
```

For example:

```
FUNCTION circle_area (radius IN REAL)
  RETURN REAL
IS
  area REAL;
BEGIN
  area := 3.141 * radius * radius;
  RETURN area;
END;
```

- Packages:

- A collection of functions and procedures can be encapsulated in a **package** that other developers can use.
- PL/SQL's `package` is weaker than `C++`'s `class`.
  $\Rightarrow$ encapsulation, but no dynamic creation.

1.8 Oracle and C

- Oracle (as does almost every dbase vendor) provides two mechanisms for programming in C and accessing the dbase:
  1. Using the Oracle Call Interface:
     - A library of C functions.
     - Functions for executing SQL, directing output and taking input from user-defined C variables.
     - Currently, Oracle supports only some loaders.
       ⇒ Use the loader `ld` in `/opt/SUNWspro/bin`.
  2. Using *precompilers*:
     - SQL-statements can be inserted in a C program.
     - The C precompiler parses the SQL blocks.
     - Precompilation allows for some compile-time optimizations.
- Oracle also has language support for Ada, Fortran and Cobol.
- First, we'll focus on the OCI SQL library.
- To create your C program that interfaces with Oracle, you will need the following includes:

```
#include <stdio.h>      /* Standard C library */
#include <ctype.h>      /* Standard C library */
#include <string.h>     /* Standard C library */

#include <oratypes.h>   /* Definitions of Oracle datatypes, constants */
#include <ocidfn.h>     /* Variable types for local data */
#include <ociapr.h>     /* Definitions of function calls */
#include <ocidem.h>     /* Constants used in demo programs. Useful */
```

1.9 Oracle OCI: Setting up a Connection

- The first thing your C program will do is to set up a connection with the Oracle server.
  ⇒ username and password need to be specified
- The library function `olog(...)` is used to set up a connection.
- The file `ORACLE_HOME/rdbms/demo/ociapr.h` contains the definition:

```
sword  olog     (struct cda_def *lda, ub1* hda,
                 text *uid, sword uidl,
                 text *pswd, sword pswdl,
                 text *conn, sword connl,
                 ub4 mode);
```

  which makes very little sense.

- Oracle defines its own data types in place of standard C data types:
  - Example: the data type `text *` is really `char *`, a character string.
  - Example: the data types `sword` and `uword` correspond to `signed int` and `unsigned int` in C.
  - Here are some of the more important ones:

```
ORACLE               C
----------------------------
text *               char *
sb1,ub1              char
sb2,ub2              signed and unsigned 2-byte int's
sb4,ub4              signed and unsigned 4-byte int's
sword, uword         signed and unsigned int's
```

- Note: most often you will use `sword` instead of `int`.
- Use these declarations for variables used in OCI calls. For example, to pass the integer `num_rows` to a call that requires `ub4`, do a cast:

```
int my_mode = 5;
ub4 mode_in_call;
mode_in_call = (ub4) my_mode;
/* Now mode_in_call can be used in the function call */
```

- These weird type names have been defined so that the definitions can vary from system to system (in the file `oratypes.h`) while keeping function definitions the same (in `ocidfn.h`).

• Every OCI program *must* have certain global variables:

```
/* Global variables */
Lda_Def lda;                        /* Logon data area */
Cda_Def cda;                        /* Cursor data area */
ub4 hda[HDA_SIZE/sizeof(ub4)];     /* Host data area */
```

- The *logon data area* is for Oracle (the OCI client) to place your connection ID, error codes relating to function calls etc.
- The *cursor data area* is for Oracle to keep track of current locations of cursors.
  Note: a cursor is a programming technique unique to dbases.
- The *host data area* is also used by Oracle for information related to your program.
- How does Oracle know the addresses of these variables?
  ⇒ you are going to pass these on to Oracle very early.
- Note: multiple lda's and cda's may be used for multiple connections. [See the documentation].

- A typical `olog` call looks like:

```
text * username = (text *) "Smith";
text * password = (text *) "S93JfiIE";
if ( olog (&lda, (ub1 *)hda, username, -1, password, -1,
         (text *) 0, -1, OCI_LM_DEF) ) {
   /* olog does not return 0 */
   printf ("Function call failed\n");
   exit (0);
}
/* Else, olog returned 0 => successful connection */
```

- Let's take a closer look:

```
if ( olog          /* Returns 0 if successful */
     (&lda,        /* Pass address of lda */
     (ub1 *)hda,   /* Pass address of hda */
     username,     /* Give username */
     -1,           /* If username is not null-terminated,
                      give length */
     password,     /* Give username */
     -1,           /* Length, if not null-terminated */
     (text *) 0,   /* For use in a networking application */
     -1,           /* Length for above */
     OCI_LM_DEF)   /* Blocking or non-blocking mode */
     ) {
```

Thus, the only "real" information is: username and password.

- Note: the C code contained the password. A more intelligent way to handle this problem:

```
text username[20];
text password[20];
printf ("username: ");   gets( (char*) username);
printf ("password: ");   gets( (char*) password);
if ( olog (&lda, (ub1 *)hda, username, -1, password, -1,
           (text *) 0, -1, OCI_LM_DEF) ) {
   printf ("Connection failed. Exiting...\n");
   exit (1);
}
```

- To close a connection, use `ologof (...)`:

```
if (ologof (&lda) ) {
  printf ("Error during log out\n");
  exit (1);
}
```

1.10 Oracle OCI: Executing SQL from C

- An SQL statement in a C program is executed with three steps:

  1. *Cursor definition*: every SQL query is associated with a cursor. Thus, a cursor is first associated with a connection using the `oopen` call:

     ```
     oopen(&cda, &lda, (text *) 0, -1, -1, (text *) 0, -1);
     ```

  2. *Parsing*: the function `oparse` is used to parse an SQL string:

     ```
     text * sqlquery = "select NAME, SSN from EMP";
     oopen(&cda, &lda, (text *) 0, -1, -1, (text *) 0, -1);
     oparse (&cda, sqlquery, -1, FALSE, VERSION_7);
     ```

  3. *Execution*: the function `oexec` is then used to execute the query:

     ```
     text * sqlquery = "select NAME, SSN from EMP";
     oopen(&cda, &lda, (text *) 0, -1, -1, (text *) 0, -1);
     oparse (&cda, sqlquery, -1, FALSE, VERSION_7);
     oexec (&cda);
     ```

- A closer look at these functions:

  1. The `oopen` function:

  ```
  oopen (&cda,       /* The cursor global variable we've defined */
         &lda,       /* The current connection identifier */
         (text *) 0, /* Remaining parameters can be ignored. */
         -1,         /* They are for compatibility with earlier */
         -1,         /* Oracle versions */
         (text *) 0,
  ```

```
        -1);
```

Thus, the `oopen` call really tells Oracle "Here's a cursor variable that I want you to associate with my current connection".

2. The `oparse` function:

```
    oparse( &cda,        /* Ptr to CDA specified by OOPEN */
            sqlquery,    /* String containing sql query*/
            (sb4) -1,    /* Length of SQL statement, if not
                            NULL terminated */
            FALSE,       /* Value 0 => parse NOW
                            Value 1 => parse later */
            VERSION_7)   /* Value 2: version info */
```

   NOTE:

   – This function causes Oracle to parse the SQL query in the string `sqlquery`.
   – It is possible to request *deferred parsing* for efficiency reasons.
   – The deferred parsing will force parsing at the time of execution.
   – The SQL statement can be a Data Definition statement (e.g., a **create table** statement) in which case it is executed if you did not defer parsing.

3. The `oexec` statement:

```
   oexec (&cda);   /* Simply specify the cursor associated
                      with the query */
```

   NOTE:

   – The `oexec` function can be used for *queries*, *updates* and *inserts*.
   – Oracle recommends using other functions for *queries* only.

- What about output from the query?

- When `oexec` is called, the query is executed and the results are placed in a temporary buffer.
- To actually obtain the data, there are two steps:
  1. Use the `odefin` call to indicate where you want the result placed (in your C variables).
  2. Use the function `ofetch` repeatedly to get one tuple at a time.

- Getting results into C variables using `odefin`:
  - Consider the following SQL statement:

    select NAME, FLT_ID
    from PASSENGER

  - Suppose that NAME is VARCHAR(50) and FLT_ID is an INTEGER.
  - In the C program, we would create the following string:

```
text * sqlquery = "select NAME, FLT_NO from PASSENGER"
```

  - This SQL statement would be parsed using `oparse`:

```
text * sqlquery = "select NAME, FLT_NO from PASSENGER";
oopen (&cda, &lda, (text *) 0, -1, -1, (text *) 0, -1);
oparse (&cda, sqlquery, -1, FALSE, VERSION_7);
```

  - Now, we want the output (NAME and FLT_NO) to be placed in C variables of our choice.
  - Accordingly, define the C variables:

```
#define PNAME_LEN 50
text pname[PNAME_LEN];
sword fltnum;
```

  - Unfortunately, Oracle requires that *three additional variables* be defined for each such attribute:

```
text pname[PNAME_LEN];     /* Passenger name */
  sb2 ind_pname;      /* An indicator code for function calls */
  ub2 retc_pname;     /* A return code - to indicate errors */
  ub2 retl_pname;     /* The length of the value returned */
sword fltnum;       /*Flight number */
  sb2 ind_fltnum;
  ub2 retc_fltnum;
  ub2 retl_fltnum;
```

Thus, for any variable `x`, you need `ind_x`, `retc_x` and `retl_x` defined.

– Next, use the `odefin` call to tell Oracle your intentions for each variable:

```
odefin (&cda, 1, (ub1*) pname, (sword) PNAME_LEN, SQLT_STR,
        -1, &ind_pname, (text *) 0, -1, -1,
        &retl_pname, &retc_pname);

odefin (&cda, 2, (ub1*) &fltnum, (sword) sizeof(int), SQLT_INT,
        -1, &ind_fltnum, (text *) 0, -1, -1,
        &retl_fltnum, &retc_fltnum);
```

– Let's look at one of these in a little more detail:

```
odefin (
 &cda,              /* The cursor associated with the query */
 1,                 /* Position of NAME in select query:
                       NAME is the *first* attribute in
                         select NAME, FLT_NO from ...
                       Use 2 for FLT_NO */
 (ub1*) pname,      /* Address of the C variable
                       - cast to ub1* */
 (sword) PNAME_LEN, /* Size of space allocated for result */
 SQLT_STR,          /* It's a string type: SQLT_STR is a
```

```
                             pre-defined constant in ocidfn.h */
       -1,                /* Not used */
       &ind_pname,        /* Did a row-fetch work? */
       (text *) 0,        /* Not used */
       -1,                /* Not used */
       -1,                /* Not used */
       &retl_pname,       /* Length of data returned to be put here */
       &retc_pname)       /* A return code */
```

– After the addresses of your C variables have been handed over to Oracle, tuples can be fetched one at a time using `ofetch`:

```
ofetch (&cda);    /* Provide the cursor identification */
```

- A complete example: consider the relations:

  ACCOUNTS (SSN, ACC_CODE, ACCNUM, BRANCHNUM, AMOUNT)
  CUSTOMER (SSN, NAME, ADDR, CITY, STATE)

  Goal: print each customer's name and balance.

  – First, `include`'s and global variables:

```
#include <stdio.h>      /* Standard C library */
#include <ctype.h>      /* Standard C library */
#include <string.h>     /* Standard C library */

#include <oratypes.h>   /* Definitions of Oracle datatypes */
#include <ocidfn.h>     /* Variable types for local data */
#include <ociapr.h>     /* Definitions of function calls */
#include <ocidem.h>     /* Constants used in demo programs */

Lda_Def lda;                      /* Logon data area */
Cda_Def cda;                      /* Cursor data area */
ub4 hda[HDA_SIZE/sizeof(ub4)];    /* Host data area */
```

– Second, let's define a function called `connect` that handles username and password:

```
void connect () {
  text username[20];
  text password[20];
  printf ("username: ");   gets( (char*) username);
  printf ("password: ");   gets( (char*) password);
  if ( olog (&lda, (ub1 *)hda, username, -1, password, -1,
           (text *) 0, -1, OCI_LM_DEF) ) {
    printf ("Connection failed. Exiting...\n");
    exit (1);
}
```

– Define the SQL statement that produces the desired result:

```
text * sqlquery
  = "select C.NAME, A.AMOUNT from ACCOUNTS A, CUSTOMER C
      where A.SSN=C.SSN";
```

– Next, define the variables in which we want the results:

```
#define CNAME_LEN 50
text cname[CNAME_LEN];     /* Customer name */
  sb2 ind_cname; ub2 retc_pname; ub2 retl_pname;
sword amount               /* Amount */
  sb2 ind_amount;  ub2 retc_amount;  ub2 retl_amount;
```

– Next, connect, define a cursor, parse the SQL statement and alias the C variables.

```
connect ();

oopen(&cda, &lda, (text *) 0, -1, -1, (text *) 0, -1);
```

```
oparse (&cda, sqlquery, -1, FALSE, VERSION_7);

odefin (&cda, 1, (ub1*) cname, (sword) CNAME_LEN, SQLT_STR,
        -1, &ind_cname, (text *) 0, -1, -1,
        &retl_cname, &retc_cname);

odefin (&cda, 2, (ub1*) &amount, (sword) sizeof(int), SQLT_INT,
        -1, &ind_amount, (text *) 0, -1, -1,
        &retl_amount, &retc_amount);
```

– Now execute and fetch resulting tuples one by one:

```
oexec (&cda);

do {
  ofetch (&cda);
  /* Check whether anything was returned
     -- look at the .rc field of cda */
  if (cda.rc == NO_DATA_FOUND) break;
  printf ("%s %d \n", cname, amount);
} while (1);
```

– Close cursors and log off:

```
oclose (&cda);
ologoff (&lda);
```

The complete code is:

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>

#include "oratypes.h"   /* Definitions of Oracle datatypes, constants */
```

```
#include "ocidfn.h"     /* Variable types for local data */
#include "ociapr.h"     /* Definitions of function calls */
#include "ocidem.h"     /* Constants used in demo programs. Useful */

#define VERSION_7 2

Lda_Def lda;                      /* Logon data area */
Cda_Def cda;                      /* Cursor data area */
ub4 hda[HDA_SIZE/sizeof(ub4)];    /* Host data area */

void connect ()
{
  text username[20];
  text password[20];

  printf ("username: ");   gets( (char*) username);
  printf ("password: ");   gets( (char*) password);
  if ( olog (&lda, (ub1 *)hda, username, -1, password, -1,
             (text *) 0, -1, OCI_LM_DEF) ) {
    printf ("Connection failed. Exiting...\n");
    exit (1);
  }
}

text * sqlquery = (text *) "select C.NAME, A.AMOUNT from simha.ACCOUNTS A, simha.CUSTOMER C where A.SSN=C.SSN";

#define CNAME_LEN 50
text cname[CNAME_LEN];     /* Customer name */
  sb2 ind_cname; ub2 retc_cname; ub2 retl_cname;
sword amount;               /* Amount */
  sb2 ind_amount;  ub2 retc_amount;  ub2 retl_amount;

int n;
```

```
void main ()
{
  connect ();

  if (oopen(&cda, &lda, (text *) 0, -1, -1, (text *) 0, -1) ) {
    printf ("oopen failed \n"); exit (0);
  }

  if (oparse (&cda, sqlquery, -1, FALSE, VERSION_7)) {
    printf ("oparse failed \n"); exit (0);
  }

  if (odefin (&cda, 1, (ub1*) cname, (sword) CNAME_LEN, SQLT_STR,
          -1, &ind_cname, (text *) 0, -1, -1,
      &retl_cname, &retc_cname)) {
    printf ("first odefin failed \n"); exit (0);
  }


  if (odefin (&cda, 2, (ub1*) &amount, (sword) sizeof(int), SQLT_INT,
             -1, &ind_amount, (text *) 0, -1, -1,
             &retl_amount, &retc_amount)) {
    printf ("second odefin failed \n"); exit (0);
  }

  if (oexec (&cda)) {
    printf ("oexec failed \n"); exit (0);
  }

  n = 0;
  do {
    ofetch (&cda);
    /* Check whether anything was returned
       -- look at the .rc field of cda */
```

```
    if ( (n > 100) || (cda.rc == NO_DATA_FOUND) ) break;
    cname[retl_cname + 1] = '\0';
    printf ("%s %d \n", cname, amount);
    n++;
  } while (1);

  printf ("%2d tuples found\n", n);

  oclose (&cda);
  ologof (&lda);

}
```

To obtain the complete code, see the file `~simha/321/oracle/test1.c`.

- Suppose we want to have the user input values to be used in queries:
  - Consider these relations:
    CUSTOMER (SSN, NAME, ADDR, CITY, STATE)
    ACCOUNTS (SSN, ACC_CODE, ACCNUM, BRANCHNUM, AMOUNT)
  - Suppose we want the user to input a customer name and obtain the amounts in his/her accounts.
  - In SQL, for customer `Ray` we would write this query as:

    select A.AMOUNT
    from CUSTOMER C, ACCOUNTS A
    where A.SSN=C.SSN and C.NAME = 'Ray';

  - However, in our C program we want to be able to *read in* the value 'Ray' from the keyboard:

```
char * cname;
printf ("Enter customer name: ");
gets ( cname );
/* We need to use cname in the SQL statement */
```

- Defining and using *placeholders* in Oracle OCI:
  - Oracle allows SQL statements to contain so-called *placeholders* or temporary variables.
  - These placeholders can be aliased to a C variable which can contain desired values.
  - The aliasing is done using either the `obndrv` or `obndrn` function calls.
  - For the above example:

```
text * sqlquery
  = (text *) "select A.AMOUNT \
              from ACCOUNTS A, CUSTOMER C \
              where A.SSN=C.SSN and C.NAME= :1";

#define CNAME_LEN 50
text cname[CNAME_LEN];     /* Customer name */
  sb2 ind_cname;
sword amount;               /* Amount */
  sb2 ind_amount;  ub2 retc_amount;  ub2 retl_amount;

/* Connect, open a cursor and parse the SQL statement */
connect ();
oopen(&cda, &lda, (text *) 0, -1, -1, (text *) 0, -1)
oparse (&cda, sqlquery, -1, FALSE, VERSION_7)

/* Use obndrn to bind cname to ":1" */
obndrn (&cda, 1, (ub1*) cname, CNAME_LEN,
        SQLT_STR, -1, &ind_cname, (text*) 0, -1, -1);

/* Alias the output */
odefin (&cda, 1, (ub1*) &amount, (sword) sizeof(int), SQLT_INT,
        -1, &ind_amount, (text *) 0, -1, -1,
        &retl_amount, &retc_amount);
```

```
 /* Now read in the customer name */
 printf ("Enter customer name: ");  gets ( (char*) cname);

oexec (&cda);

/* Fetch tuples from result */
do {
  ofetch (&cda);
  if (cda.rc == NO_DATA_FOUND) break;
  printf ("%s %d \n", cname, amount);
} while (1);
```

– Note that an SQL placeholder is a string *preceded* by a colon.

– SQL placeholders are of two types:

1. *Numeric placeholders*: these are integers (starting from 1), e.g.,

```
text * sqlquery
  = (text *) "select C.NAME \
              from CUSTOMER C \
              where C.NAME= :1 and C.CITY = :2";
```

   Numeric placeholders are always *bound* using the `obndrn` call, e.g., for the above example:

```
#define CNAME_LEN 50
#define TOWN_LEN 50
text cname[CNAME_LEN];
text ccity[CITY_LEN];
obndrn (&cda, 1, (ub1*) cname, CNAME_LEN,
        SQLT_STR, -1, &ind_cname, (text*) 0, -1, -1);

obndrn (&cda, 2, (ub1*) ccity, CITY_LEN,
        SQLT_STR, -1, &ind_ccity, (text*) 0, -1, -1);
```

2. *String placeholders*: these are strings used for identification (and have nothing to do with C variables).

```
text * sqlquery
  = (text *) "select C.NAME \
              from CUSTOMER C \
              where C.NAME= :cname and C.CITY = :town";
```

String placeholders are always bound using `obndrv`:

```
#define CNAME_LEN 50
#define TOWN_LEN 50
text cname[CNAME_LEN];
text ccity[CITY_LEN];
obndrv (&cda, (ub1*) ":cname", -1, (ub1*) cname, CNAME_LEN,
         SQLT_STR, -1, &ind_cname, (text*) 0, -1, -1);

obndrv (&cda, (ub1*) ":town", -1, (ub1*) ccity, CITY_LEN,
         SQLT_STR, -1, &ind_ccity, (text*) 0, -1, -1);
```

– Let's take a look at `obndrv`:

```
obndrv (
  &cda,                 /* The cursor reference */
  (ub1*) ":city",       /* Placeholder string */
  -1,                   /* If not null-delimited */
  (ub1*) ccity,         /* Address of C variable */
  CITY_LEN,             /* Space allocated */
  SQLT_STR,             /* Variable type */
  -1,                   /* Not used */
  &ind_ccity,           /* For return code */
  (text*) 0, -1, -1);   /* Not used */
```

– An example of using placeholders can be found in `~simha/321/oracle/test2.c`.

- OCI summary:

- Oracle OCI is a library of C functions that let you pass in SQL statements for execution.
- Use `olog, ologof` to set up a connection.
- Use `oopen` and `oclose` to open and close cursors.
- Use `odefin` to SQL results into C variables.
- Use `obndrv` and `obndrn` to use C variables as part of SQL statements.
- Use `oparse` and `oexec` to parse and execute a query.
- Use `ofetch` to retrieve one tuple at a time from the results.
- Other functions are available – see manual.

1.11 Embedded SQL: Using the Oracle C Precompiler

- An alternative to using an SQL library is to use a *precompiler* and *embed* SQL in a C program.
- The precompiler is a *translator* provided by Oracle that converts a C program with embedded SQL into a C program with function calls.
- Embedded SQL makes it easy to declare C variables for input and output.
- Embedded SQL allows SQL's datatypes to be declared like C variables, e.g.,

```
varchar cname[50];   /* An SQL type */
char custname[50];   /* Standard C type */
```

- Embedded SQL makes it easy to declare and fetch from cursors, e.g.,

```
EXEC SQL DECLARE cust_cursor CURSOR FOR
  SELECT C.NAME, A.AMOUNT
  FROM ACCOUNTS A, CUSTOMER C
  WHERE A.SSN=C.SSN and C.NAME = :custname;

EXEC SQL OPEN cust_cursor;
```

- Let's examine a complete program:

  This program reads in a customer's name and prints out the amounts owned by this customer.

```
#include <stdio.h>
#include <string.h>
#include "sqlca.h"   /* You need this include */

#define     UNAME_LEN      20
#define     PWD_LEN        20
varchar username[UNAME_LEN];   /* Note the varchar data type */
varchar password[PWD_LEN];
char usern[UNAME_LEN];
char passw[PWD_LEN];

/* We will obtain these attributes from a query */
varchar cname[50];   /* Corresponds to SQL's VARCHAR */
int amount;          /* Corresponds to SQL's INTEGER */

char custname[50];

/* Must declare error handling function. */
void sql_error();

main()
{

    printf ("username: ");   gets( (char*) usern);
    printf ("password: ");   gets( (char*) passw);
    /* Connect to ORACLE--
     * Copy the username into the VARCHAR.
     */
    strncpy((char *) username.arr, usern, UNAME_LEN);

    /* Set the length component of the VARCHAR. */
    username.len = strlen((char *) username.arr);
```

```
/* NOTE the structure of varchar: fields .arr and .len */

/* Copy the password. */
strncpy((char *) password.arr, passw, PWD_LEN);
password.len = strlen((char *) password.arr);

/* Register sql_error() as the error handler. */
EXEC SQL WHENEVER SQLERROR DO sql_error("ORACLE error--\n");

/* Connect to ORACLE.  Program will call sql_error()
 * if an error occurs when connecting to the default database.
 */
EXEC SQL CONNECT :username IDENTIFIED BY :password;

printf("\nConnected to ORACLE as user: %s\n", username.arr);

printf("\nEnter customer name: ");
gets(custname);
printf ("Customer name entered: %s\n", custname);

/* NOTE the use of a C variable as placeholder */
EXEC SQL DECLARE cust_cursor CURSOR FOR
  SELECT C.NAME, A.AMOUNT
  FROM simha.ACCOUNTS A, simha.CUSTOMER C
  WHERE A.SSN=C.SSN and C.NAME = :custname;

EXEC SQL OPEN cust_cursor;

printf ("Amounts owned by Customer %s: \n", custname);

for (;;) {
  EXEC SQL WHENEVER NOT FOUND DO break;
  EXEC SQL FETCH cust_cursor
    INTO :cname, :amount;
```

```
      /* NOTE the availability of Oracle datatypes for use in C */
      printf ("%s %4d\n", cname.arr, amount);
    }

    EXEC SQL CLOSE cust_cursor;

    EXEC SQL COMMIT WORK RELEASE;
    exit(0);
}


void sql_error (char *msg)
{
    char err_msg[128];
    size_t buf_len, msg_len;

    EXEC SQL WHENEVER SQLERROR CONTINUE;

    printf("\n%s\n", msg);
    buf_len = sizeof (err_msg);
    sqlglm(err_msg, &buf_len, &msg_len);
    printf("%.*s\n", msg_len, err_msg);

    EXEC SQL ROLLBACK RELEASE;
    exit(1);
}
```

- NOTE:
  - While not always necessary, it's best to enclose those variables that store parts of tuples in an embedded SQL block:

```
EXEC SQL BEGIN DECLARE SECTION;
  varchar cname[50];
```

```
  int amount;
 EXEC SQL END DECLARE SECTION;
```

– Pointers to character arrays can be used instead of declaring a fixed size of type `varchar`:

```
EXEC SQL BEGIN DECLARE SECTION;
  char *cname;
  int amount;
EXEC SQL END DECLARE SECTION;

char *custname;
...
custname = (char *) malloc (50*sizeof(char));
...
EXEC SQL DECLARE cust_cursor CURSOR FOR
  SELECT C.NAME, A.AMOUNT
  FROM simha.ACCOUNTS A, simha.CUSTOMER C
  WHERE A.SSN=C.SSN and C.NAME = :custname;
  EXEC SQL OPEN cust_cursor;

for (;;) {
  EXEC SQL WHENEVER NOT FOUND DO break;
  EXEC SQL FETCH cust_cursor
    INTO :cname, :amount;
  printf ("%4d\n", amount);
}
...
```

However, it's best to use `varchar` where possible.

– It is often inefficient to fetch only a tuple at a time.

  * Consider the following relation:
    EMP (NAME, SSN, AGE)

    with millions of tuples.

∗ Each tuple fetch may occur across a network (client-server mode)
  ⇒ very time-consuming.

∗ It's better to retrieve *batches* of tuples.

∗ First, declare an array (of your desired batch size) of variables to receive the output of the SQL statement:

```
EXEC SQL BEGIN DECLARE SECTION;
  varchar ename[20][50];  /* Batch size 20, ename size is 50 */
  varchar ssn[20][12];
  int age[20];
EXEC SQL END DECLARE SECTION;
```

NOTE: arrays of pointers and multidimensional arrays are not allowed.

∗ Next, declare the cursor:

```
EXEC SQL DECLARE emp_cursor FOR
  SELECT E.NAME, E.SSN, E.AGE
  FROM EMP E;
```

∗ Each fetch from the cursor will retrieve a batch:

```
for (;;) {
  EXEC SQL WHENEVER NOT FOUND DO break;
  EXEC SQL FETCH emp_cursor;
}
```

∗ It is possible that fewer than 20 items are retrieved
  ⇒ could be near end of table.

∗ The precompiler-defined variable `sqlca.sqlerrd[2]` keeps track of the *number of tuples returned so far.*

∗ Thus, the following method tells you how many rows were returned:

```
n = 0;
for (;;) {
  EXEC SQL WHENEVER NOT FOUND DO break;
```

```
    EXEC SQL FETCH emp_cursor;
    rows_returned = sqlca.sqlerrd[2] - n;
    n = sqlca.sqlerrd[2];
  }
```

  - Note: batches can be used for *insert*'s, *update*'s and *delete*'s as well.

- Compilation and execution:
  - The `C/C++` precompiler in Oracle is called `proc` (for Pro-C).
  - The suffix used for embedded SQL (in C) programs is `.pc`, for example, `testpc1.pc`.
  - Now, as part of compilation, a C file will be generated. Thus, if you have a C file called `test1.c` then DO NOT name your Pro-C file test1.pc
    ⇒ compilation will overwrite your test1.c file.
  - Consider the file `testpc1.pc`.
  - You can compile this file as follows:

    ```
    % proc testpc1.pc
    ```

    However, you are likely to get linking errors.
  - Oracle recommends that you use their pre-defined *makefile* for compilation.
  - Copy over their `proc.mk` makefile and use it as:

    ```
    % make -f proc.mk EXE=testpc1 OBJ=testpc1.o
    ```

  - Observe that several files are produced as the result of compilation:
    * `testpc1.c` – this is a plain C file which has the embedded SQL replaced by incomprehsible C code.
    * `testpc1.o` – an object file.
    * `testpc1` – an executable.

– Unfortunately, you don't get much help with errors.

  Two kinds of compilation errors are produced:

  1. Pre-compiler errors: these are likely errors in the embedded SQL.
  2. C compiler errors: errors found in the "intermediate" C file `testpc1.c`.

- What does the mysterious "intermediate" file look like?

  For example, the declaration

```
varchar cname[50];
```

  becomes

```
/* varchar cname[50]; */
struct { unsigned short len; unsigned char arr[50]; } cname;
```

  Similarly, the statement:

```
EXEC SQL OPEN cust_cursor
```

  becomes

```
struct sqlexd sqlstm;
sqlstm.sqlvsn = 8;
sqlstm.arrsiz = 3;
sqlstm.stmt = sq0002;
sqlstm.iters = (unsigned int  )1;
sqlstm.offset = (unsigned int  )28;
...
sqlcxt((void **)0, &sqlctx, &sqlstm, &sqlfpn);
if (sqlca.sqlcode < 0) sql_error("ORACLE error--\n");
```

Thus, it has been translated to the function call `sqlcxt`.

- Embedded SQL in C: a summary
  - SQL statements can be embedded in a C program.
  - C variables can be used to determine values in SQL statements.
  - C variables can be used to receive the results of SQL queries.
  - A precompiler translates the source into a regular C program, which is then compiled by a regular C compiler.

1.12 Useful SQL Functions in Oracle

- Oracle provides a number of useful functions to manipulate various data types.
- Type `help functions` in `sqlplus` for descriptions.
- *Numeric functions*:

  Consider the following relation:

  ```
  create table PATIENT ( NAME     VARCHAR(50),
                         AGE      INTEGER,
                         WEIGHT   FLOAT,
                         HEIGHT   FLOAT,
                         BALANCE  NUMBER(8,2) );
  ```

  The following numeric functions can be used: `ceil, floor, mod, power, round, sign, sqrt`

  – Example:

  ```
  select  round (P.WEIGHT, 2)
  from    PATIENT P;
  ```

  – Example:

  ```
  select  P.NAME, sign (P.BALANCE)
  from    PATIENT P;
  ```

  – Example:

  ```
  select  P.NAME, P.WEIGHT, P.HEIGHT
  from    PATIENT P
  where   floor (P.WEIGHT) > 10;
  ```

- *Character functions*:

  The following functions operate on strings and return strings:

– `initcap`: capitalize first letter of string, e.g.,

```
select  initcap (E.NAME)
from    EMP E;
```

– `replace(strA, strB, strC)`: Replace every occurence of string `strB` with `strC` in the string `strA`, e.g.,

```
select  replace (E.POSITION,'man','person')
from    EMP E
where   E.POSITION = 'Chairman';
```

– `soundex`: for phonetic searches:

```
select  E.NAME
from    EMP E
where   soundex (E.NAME) = soundex ('Meyer');
```

(Should match: Maier, Meier, Mayer, Meyer).

– Other functions of interest: `lower, substr, length`.

- *Date functions*:

  Several date-related functions are useful:

  – `sysdate` – current date and time, e.g.,

```
select  L.NAME, L.BOOK
from    LIBRARY_BOOKS L
where   L.DUE_DATE < sysdate
```

  – `last_day` – last day of the month, e.g.,

```
select  L.NAME, L.BOOK
from    LIBRARY_BOOKS L
where   last_day (L.DUE_DATE) < sysdate;
```

  – `months_between` – difference in months betwen 2 dates, e.g.,

```
select  L.NAME, L.BOOK
from    LIBRARY_BOOKS L
where   months_between (L.DUE_DATE), sysdate) > 3;
```

- Conversion functions: to convert from one data type to another.

– `to_char` – converts from *any* data type to a character string, e.g.

select M.TITLE, M.YEAR
from MOVIES M
where M.TITLE = **to_char** (10)

– `to_date` – converts *character* data in the specified format to a date, e.g.,

select P.NAME
from PATIENTS P
where P.BIRTHDAY = **to_date** (24-Jan-66,'DD)

The function **to_date** figures out the *day of the week* for 24-Jan-66 for use in the comparison.