# Chapter 5

# Lecture (Lec 6 & 7) for Week 4

Introduction to PL SQL.

**A few points about PL SQL:**

- PL/SQL lets you write code once and deploy it in the database nearest the data. PL/SQL can simplify application development, optimize execution, and improve resource utilization in the database.
- The language is a case-insensitive programming language, like SQL.
- **History:** PL/SQL was developed by modeling concepts of structured programming, static data typing, modularity, exception management, and parallel (concurrent) processing found in the *Ada programming language*. The Ada programming language, developed for the United States Department of Defense, *was designed to support military real-time and safety-critical embedded systems, such as those in airplanes and missiles.* The Ada programming language *borrowed significant syntax from the Pascal programming language*, including the assignment and comparison operators and the single-quote delimiters.

### 5.0.1 Block

PL/SQL is a blocked programming language. Program units can be *named or unnamed* blocks. Unnamed blocks are known as *anonymous* blocks and are labeled so throughout the book. The PL/SQL coding style differs from that of the C, C++, and Java programming languages. For example, curly braces do not delimit blocks in PL/SQL.

Anonymous-block programs are effective in some situations. You typically use anonymous blocks when building scripts to seed data or perform one-time processing activities. They are also effective when you want to nest activity in another PL/SQL blocks execution section. The basic anonymous-block structure must contain an execution section. You can also put optional declaration and exception sections in anonymous blocks. The following illustrates an anonymous-block prototype:

```
 [DECLARE]
declaration_statements
```

```
BEGIN
execution_statements
[EXCEPTION]
exception_handling_statements
END;
/
```

Lets write out first block that will do nothing:

```
 BEGIN
NULL;
END;
/
```

**Hello World:**

```
 SET SERVEROUTPUT ON SIZE 1000000
BEGIN
dbms_output.put_line('Hello World.');
END;
/
```

**scanf:**

```
 --scanf--

DECLARE
my_var VARCHAR2(30);
BEGIN
my_var := '&i';
dbms_output.put_line('Hello '|| my_var );
END;
```

### 5.0.2 Variables, Assignments, and Operators

**Delimiters**: Lexical delimiters are symbols or symbol sets.

For complete set see table 3-1. Here are some important symbols frequently used in pl sql programming.

- **:= Assignment**

  The assignment operator is a colon immediately followed by an equal symbol. It is the only assignment operator in the language. You assign a right operand to a left operand, like a := b + c;

- **. Association**

  The component selector is a period, and it glues references together, for example, a schema and a table, a package and a function, or an object and a member method. Component selectors are also used to link cursors and cursor attributes (columns). The following are some prototype examples:

```
schema_name.table_name
package_name.function_name
object_name.member_method_name
cursor_name.cursor_attribute
object_name.nested_object_name.object_attribute
These are referenced in subsequent chapters throughout this book.
```

- **@ Association.** The remote access indicator lets you access a remote database through database links.
- **= Comparison.**
- **<> != Comparison** Not equal to.
- **– Delimiter** In-line comment.
- **/* Delimiter */** Multi-line comment.
- **" Delimiter.**

```
 quoted identifier delimiter is a double quote. It lets you access ta
in case-sensitive fashion from the database catalog. This is required
created database catalog objects in case-sensitive fashion. You can d
Oracle 10g forward.
For example, you create a case-sensitive table or column by using quo
delimiters:
CREATE TABLE "Demo"
("Demo_ID" NUMBER
, demo_value VARCHAR2(10));
You insert a row by using the following quote-delimited syntax:
INSERT INTO "Demo1" VALUES
(1,'One Line ONLY.');#
```

## 5.1 Data-types

*See Figure 3.5 for details.*

### 5.1.1 Scalar Datatypes

Scalar datatypes use the following prototype inside the declaration block of your programs:

```
variable_name datatype [NOT NULL] [:= literal_value];
```

1. **Boolean.** The BOOLEAN datatype has three possible values: TRUE, FALSE, and NULL.

   ```
   var1 BOOLEAN; -- Implicitly assigned a null value.
   var2 BOOLEAN NOT NULL := TRUE; -- Explicitly assigned a TRUE value.
   var3 BOOLEAN NOT NULL := FALSE; -- Explicitly assigned a FALSE valu
   ```

   There is little need to subtype a BOOLEAN datatype, but you can do it. The subtyping syntax is:

   ```
   SUBTYPE booked IS BOOLEAN;
   ```

   This creates a subtype BOOKED that is an unconstrained BOOLEAN datatype. You may find this useful when you need a second name for a BOOLEAN datatype, but generally subtyping a Boolean is not very useful.

2. **Characters and Strings.**

   The VARCHAR2 datatype stores variable-length character strings. When you create a table with a VARCHAR2 column, you specify a maximum string length (in bytes or characters) between 1 and 4000 bytes for the VARCHAR2 column. For each row, Oracle Database stores each value in the column as a variable-length field unless a value exceeds the column's maximum length, in which case Oracle Database returns an error. Using VARCHAR2 and VARCHAR saves on space used by the table.

   **Difference between Char and Varchar2:** The following program illustrates the memory allocation differences between the CHAR and VARCHAR2 datatypes:

   ```
    DECLARE
   c CHAR(2000) := 'hello';
   v VARCHAR2(32767) := 'hello';
   BEGIN
   dbms_output.put_line('The len of c is :'||LENGTH(c)||);
   dbms_output.put_line('The len of v is :'||LENGTH(v)||);
    END;
   /
   ```

   Output:

```
The len of c is :2000
The len of v is : 5
```

The output shows that a CHAR variable sets the *allocated memory size when defined*. The allocated memory can exceed what is required to manage the value in the variable. The output also shows that the VARCHAR2 variable *dynamically allocates* only the required memory to host its value.

**More on VARCHAR2:** Globalization support allows the use of various character sets for the character datatypes. Globalization support lets you process *single-byte* and *multibyte* character data and convert between character sets. Client sessions can use client character sets that are different from the database character set.

The length semantics of character datatypes can be measured in *bytes* or *characters*.

- **Byte semantics** treat strings as a sequence of bytes. This is the default for character datatypes.
- **Character semantics** treat strings as a sequence of characters. A character is technically a codepoint of the database character set.

For *single byte character* sets, columns defined in character semantics are basically the same as those defined in byte semantics. *Character semantics are useful for defining varying-width multibyte strings*; it reduces the complexity when defining the actual length requirements for data storage. *For example*, in a Unicode database (UTF8), you must define a VARCHAR2 column that can store up to five Chinese characters together with five English characters. In byte semantics, this would require (5*3 bytes) + (1*5 bytes) = 20 bytes; in character semantics, the column would require 10 characters.

Lets look at the example below:

```
 var1 VARCHAR2(100); -- Implicitly sized at 100 byte.
var2 VARCHAR2(100 BYTE); -- Explicitly sized at 100 byte.
var3 VARCHAR2(100 CHAR); -- Explicitly sized at 100 character.
```

When you use character space allocation, the maximum size changes, depending on the character set of your database. Some character sets use *two or three bytes to store characters*. You divide 32,767 by the number of bytes required per character, which means the maximum for a VARCHAR2 is 16,383 for a two-byte character set and 10,922 for a three-byte character set.

3. **Date:** Already discussed. See interval and TIMESTAMP.

4. **NUMBER.** It can store numbers in the range of 1.0E-130 (1 times 10 raised to the negative 130th power) to 1.0E126 (1 times 10 raised to the 126th power).

   The following is the prototype for declaring a fixed-point NUMBER datatype:

   ```
   NUMBER[(precision, [scale])] [NOT NULL]
   ```

5. **Large Objects (LOBs)** ( Details will be covered later. ) Large objects (LOBs) provide you with four datatypes: BFILE, BLOB, CLOB, and NCLOB. The BFILE is a datatype that points to an external file, which limits its maximum size to 4 gigabytes. The BLOB, CLOB and NCLOB are internally managed types, and their maximum size is 8 to 128 terabytes, depending on the`db_block_size` parameter value.

### 5.1.2 Composite Datatypes

There are two composite generalized datatypes: records and collections. (Will be covered next).

Just have look at the following example for Record.

```
 DECLARE
TYPE demo_record_type IS RECORD
( id NUMBER DEFAULT 1
, value VARCHAR2(10) := 'One');
demo DEMO_RECORD_TYPE;
BEGIN
dbms_output.put_line('['||demo.id||']['||demo.value||']');
END;
/
```

# Chapter 6

# Lecture (Lec 7 to 9) for Week 3

## 6.1 Control Structure

### 6.1.1 Conditional statements

1. **IF Statements.** IF statements evaluate a condition. The condition can be any comparison expression, or set of comparison expressions that evaluates to a logical true or false.

   **Example:**

```
DECLARE
X NUMBER;
BEGIN
X:=10;
IF (X = 0) THEN
 dbms_output.put_line('The value of x is 0 ');
ELSIF(X between 1 and 10) THEN
dbms_output.put_line('The value of x is between 1 and 10 ');
ELSE
dbms_output.put_line('The value of x is greater than 10 ');
END IF;
END;
```

   **Example in Real Scenario:**

```
  DECLARE
CGPA NUMBER;
X NUMBER :=034403;
BEGIN

SELECT MAX(CGPA) INTO CGPA
```

```
FROM STUDENTS
WHERE ID=X;

IF (CGPA >3.78) THEN
 dbms_output.put_line('Brilliant');
ELSIF(CGPA between 3.5 and 3.78) THEN
dbms_output.put_line('Mid Level');
ELSE
dbms_output.put_line('Poor');
END IF;
END;
```

2. **Simple CASE Statements.** The simple CASE statement sets a selector that is any PL/SQL datatype except a BLOB, BFILE, or composite type.

   **Example:**

```
  DECLARE
selector NUMBER := 1;
BEGIN
CASE selector
WHEN 0 THEN
dbms_output.put_line('Case 0!');
WHEN 1 THEN
dbms_output.put_line('Case 1!');
ELSE
dbms_output.put_line('No match!');
END CASE;
END;
/
```

   **A more practical Example:** Normally it is used in aid of function or procedure. In the following example it uses a number of procedures ( that does some specific task almost like function, without having any return statement).

```
CASE employee_type
WHEN 'S' THEN
award_salary_bonus(employee_id);
WHEN 'H' THEN
award_hourly_bonus(employee_id);
WHEN 'C' THEN
award_commissioned_bonus(employee_id);
```

```
ELSE
NULL; ---do nothing
END CASE;
```

3. **Searched CASE Statements**

   Lets look at the example: (this is one step advancement in SQL). Note: It can be used inside your PL SQL code with **into** clause.

```
  SELECT name, ID,
(CASE
  WHEN salary < 1000 THEN 'Low'
  WHEN salary BETWEEN 1000 AND 3000 THEN 'Medium'
  WHEN salary > 3000 THEN 'High'
  ELSE 'N/A'
      END) salary
    FROM emp
    ORDER BY name;
```

**CASE can be used for selecting the value of a parameter dynamically:**

```
give_bonus(employee_id,x);
```

can be written more dynamically as follows:

```
give_bonus(employee_id,
CASE
WHEN salary >= 10000 AND salary <= 20000 THEN 1500
WHEN salary > 20000 AND salary <= 40000 THEN 1000
WHEN salary > 40000 THEN 500
ELSE 0
END);
```

### 6.1.2 LOOP

1. **LOOP and EXIT Statements** Simple loops are explicit block structures. A simple loop starts and ends with the LOOP reserved word. An EXIT statement or an EXIT WHEN statement is required to break the loop.

   **Example: LOOP EXIT:**

```
  DECLARE
   x number := 10;
BEGIN
   LOOP
      dbms_output.put_line(x);
      x := x + 10;
      IF x > 50 THEN
         exit;
      END IF;
   END LOOP;
   -- after exit, control resumes here
   dbms_output.put_line('After Exit x is: ' || x);
END;
/

SQL> /
10
20
30
40
50
After Exit x is: 60

PL/SQL procedure successfully completed.
```

**Example: LOOP EXIT WHEN**

```
  DECLARE
   x number := 10;
BEGIN
   LOOP
      dbms_output.put_line(x);
      x := x + 10;
      exit WHEN x > 50;
   END LOOP;
   -- after exit, control resumes here
   dbms_output.put_line('After Exit x is: ' || x);
END;
/
```

2. **FOR Loop.** A FOR LOOP is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

**Syntax:**

```
  FOR counter IN initial_value .. final_value LOOP
   sequence_of_statements;
END LOOP;
```

*Few Points:*

- After the body of the for loop executes, the value of the counter variable is increased or decreased.
- The `initial_value` and `final_value` of the loop variable or counter can be literals, variables, or expressions but must evaluate to numbers. Otherwise, `PL/SQL` raises the predefined exception `VALUE_ERROR`.
- The `initial_value` need not to be 1; however, the loop counter increment (or decrement) must be 1.
- `PL/SQL` allows determine the loop range dynamically at run time.

**Example:**

```
 DECLARE
   a number(2);
BEGIN
   FOR a in 10 .. 20 LOOP
       dbms_output.put_line('value of a: ' || a);
  END LOOP;
END;
/

OUTPUT:

value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
value of a: 20
```

```
PL/SQL procedure successfully completed.
```

**Reverse FOR LOOP:**By default, iteration proceeds from the initial value to the final value, generally upward from the lower bound to the higher bound. You can reverse this order by using the REVERSE keyword. In such case, iteration proceeds the other way. After each iteration, the loop counter is decremented.

However, you must write the range bounds *in ascending (not descending) order*.

**Example:**

```
  DECLARE
   a number(2) ;
BEGIN
   FOR a IN REVERSE 10 .. 20 LOOP
      dbms_output.put_line('value of a: ' || a);
   END LOOP;
END;
/

OUTPUT:

value of a: 20
value of a: 19
value of a: 18
value of a: 17
value of a: 16
value of a: 15
value of a: 14
value of a: 13
value of a: 12
value of a: 11
value of a: 10

PL/SQL procedure successfully completed.
```

3. **WHILE Loop.** A WHILE LOOP statement in PL/SQL programming language repeatedly executes a target statement as long as a given condition is true.

   **Syntax:**

```
 WHILE condition LOOP
```

```
   sequence_of_statements
END LOOP;
```

**Example:**

```
 DECLARE
   a number(2) := 10;
BEGIN
   WHILE a < 20 LOOP
      dbms_output.put_line('value of a: ' || a);
      a := a + 1;
   END LOOP;
END;
/

OUTPUT:

value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19

PL/SQL procedure successfully completed.
```

**Example:**

```
   DECLARE
   a number(2) := 10;
BEGIN
   WHILE a < 20 LOOP
      dbms_output.put_line('value of a: ' || a);
      a := a + 1;
   END LOOP;
END;
/
```

4. **Cursor.** Will be covered after function and procedure.

# Chapter 7

# Lecture (Lec 10 & 11 for Week 6

## 7.1 Functions and Procedures

*Motivation for Functions and Procedures:* **Modular Code.** Modularization is the process by which you break up large blocks of code into smaller pieces (modules) that can be called by other modules. Modularization of code is analogous to normalization of data, with many of the same benefits and a few additional advantages. With modularization, your code becomes:

- **More reusable.** By breaking up a large program or entire application into individual components that plug-and-play together, you will usually find that many modules are used by more than one other program in your current application. Designed properly, these utility programs could even be of use in other applications!
- **More manageable.** Which would you rather debug: a 1,000-line program or five individual 200-line programs that call each other as needed? Our minds work better when we can focus on smaller tasks. You can also test and debug on a per-program scale (called unit testing) before individual modules are combined for a more complicated integration test.
- **More readable.** Modules have names, and names describe behavior. The more you move or hide your code behind a programmatic interface, the easier it is to read and understand what that program is doing. Modularization helps you focus on the big picture rather than on the individual executable statements. You might even end up with that most elusive kind of software: self-documenting code.
- **More reliable.** The code you produce will have fewer errors. The errors you do find will be easier to fix because they will be isolated within a module. In addition, your code will be easier to maintain because there is less of it and it is more readable.

*Forms of Modularization:* PL/SQL offers the following structures that modularize your code in different ways:

- **Procedure.** A program that performs one or more actions and is called as an executable PL/SQL statement. You can pass information into and out of a procedure through its parameter list.

- **Function.** A program that returns data through its RETURN clause, and is used just like a PL/SQL expression. You can pass information into a function through its parameter list.
- **Database trigger.** A set of commands that *are triggered to execute (e.g., log in, modify a row in a table, execute a DDL statement) when an event occurs* in the database.
- **Package.** A named collection of procedures, functions, types, and variables. A package is not really a module (its more of a meta-module), but it is so closely related that.

### 7.1.1 Procedures and Function

**Procedure:**A procedure is a module that performs one or more actions. Because a procedure call is a standalone executable statement in PL/SQL, a PL/SQL block could consist of nothing more than a single call to a procedure. Procedures are key building blocks of modular code, allowing you to both consolidate and reuse your program logic.

**Syntax:**

```
   [CREATE [OR REPLACE]]
PROCEDURE procedure_name[(parameter[, parameter]...)]
   [AUTHID {DEFINER | CURRENT_USER}] {IS | AS}
   [PRAGMA AUTONOMOUS_TRANSACTION;]
   [local declarations]
BEGIN
   executable statements
[EXCEPTION
   exception handlers]
END [name];
```

**Function:** A function is a module that returns data through its RETURN clause, rather than in an OUT or IN OUT argument (We will see what it means). Unlike a procedure call, which is a standalone executable statement, a call to a function can exist only as part of an executable statement, such as an element in an expression or the value assigned as the default in a declaration of a variable.

**Syntax**

```
   [CREATE [OR REPLACE]]
FUNCTION function_name[(parameter[, parameter]...)]
RETURN RETURN_TYPE
   [AUTHID {DEFINER | CURRENT_USER}] {IS | AS}
   [PRAGMA AUTONOMOUS_TRANSACTION;]
   [local declarations]
BEGIN
   executable statements
```

```
[EXCEPTION
   exception handlers]

   RETURN STATEMENT;
END [name];
```

**Common in Procedure and Function:**

- Has a name.
- Can take parameters, and can return values.
- Is stored in the data dictionary.
- Can be called by many users.

*Note:* The term *stored procedure* is sometimes used generically for both stored procedures and stored functions. The only difference between procedures and functions is that functions always return a single value to the caller, while procedures do not return a value to the caller.

#### 7.1.1.1 Parameters for Procedures and Functions

Parameter modes define the behavior of formal parameters. The three parameter modes, **IN (the default), OUT, and IN OUT**, can be used with any subprogram. However, avoid using the OUT and IN OUT modes with functions.

- **IN:** The value of the actual parameter is passed into the procedure when the procedure is invoked. Inside the procedure, the formal parameter acts like a PL/SQL constantit is considered *read-only* and cannot be changed.
- **OUT:** Any value the actual parameter has when the procedure is called is ignored. Inside the procedure, *the formal parameter acts like an uninitialized PL/SQL variable and thus has a value of NULL*. It can be read from and written to.
- **INOUT:** This mode is a combination of IN and OUT.

**Formal Parameters:** When you declare a parameter, however, you must leave out the constraining part of the declaration.

**Datatype indicator.** 1. Using the `%TYPE Attribute` 2. Using the `%ROWTYPE Attribute`

```
   DECLARE
   emprec    employees_temp%ROWTYPE;
   id emp.id%TYPE;
BEGIN
   emprec.id := NULL;  -- this works, null constraint is not inherited
-- emprec.id := 10000002; -- invalid, number precision too large
```

```
   emprec.dept := 50; -- this works, check constraint is not inherited
-- the default value is not inherited in the following
   DBMS_OUTPUT.PUT_LINE('emprec.deptname: ' || emprec.deptname);
END;
/
```

**Example: Procedure with IN and OUT parameter**

```
   CREATE OR REPLACE PROCEDURE PROC1(ID IN NUMBER, SALARY OUT NUMBER)
AS BEGIN
SELECT MAX(SALARY) INTO SALARY
FROM EMP
WHERE ID = ID;
END;
/
---now call it from annonymous block--
DECLARE
amount NUMBER;
BEGIN
PROC1(101, amount);
dbms_output.put_line(amount);
END;
```

**Parameter Positions:**

- **Positional Notation:** You use positional notation to call the function as follows:

```
  BEGIN
dbms_output.put_line(add_three_numbers(3,4,5));
END;
```

- **Named Notation:** You call the function using named notation by:

```
  BEGIN
dbms_output.put_line(add_three_numbers(c => 4,b => 5,c => 3));
END;
```

- **Mixed Notation:** You call the function by a mix of both positional and named notation by:

```
  BEGIN
dbms_output.put_line(add_three_numbers(3,c => 4,b => 5));
END;
```

*Note: There is a restriction on mixed notation. All positional notation actual parameters must occur first and in the same order as they are defined by the function signature.*

#### 7.1.1.2 Calling Procedure and Function

**Procedure:**For procedure you cannot call as part of your SELECT statement (since it does not return anything).

```
BEGIN
apply_discount( new_company_id, 0.15 ); -- 15% discount
END;
```

Without parameters:

```
display_store_summary;
display_store_summary();
```

**Function:** You can call it in any SELECT statement like:

```
SELECT ID, NAME, GET_GPA(ID) RESULT
FROM STUDENTS;
```

Inside the PL SQL block:

```
s number:=0;
.....

s:=get_bonus(ID,category);
```

#### 7.1.1.3 Default values in parameters

It is possible to initialize subprogram parameters just like any variable. Look at the following example:

```
CREATE OR REPLACE PROCEDURE astrology_reading
(sign_in IN VARCHAR2 := 'LIBRA',
born_at_in IN DATE DEFAULT SYSDATE) IS

.......

......
```

Now calls can be made as follows:

```
astrology_reading ('SCORPIO',
TO_DATE ('12-24-2009 17:56:10', 'MM-DD-YYYY HH24:MI:SS'));
astrology_reading ('SCORPIO');
astrology_reading;
astrology_reading();
```

The **first call** specifies both parameters explicitly. In the **second call**, only the first actual parameter is included, so `born_at_in` is set to the current date and time. In the **third call**, no parameters are specified, so the parentheses was omitted (the same goes for the final)

*Subprogram Overloading* is also possible. (see the text book for details)

# Chapter 8

# Lecture (Lec 11 & 12 for Week 7

## 8.1 Cursor

In response to any DML statement the database creates a memory area, known as *context area*, for processing an SQL statement, which contains all information needed for processing the statement, for example, number of rows processed, etc.

A cursor is a pointer to this context area. PL/SQL controls the context area through a cursor. A cursor holds the rows (one or more) returned by a SQL statement. The set of rows the cursor holds is referred to as the *active set*.

You can name a cursor so that it could be referred to in a program to fetch and process the rows returned by the SQL statement, one at a time. There are two types of cursors:

- Implicit cursors.
- Explicit cursors.

### 8.1.1 Implicit Cursor

Implicit cursors are automatically created by Oracle whenever an SQL statement is executed, when there is no explicit cursor for the statement. Programmers cannot control the implicit cursors and the information in it.

Whenever a DML statement (INSERT, UPDATE and DELETE) is issued, an implicit cursor is associated with this statement. For INSERT operations, the cursor holds the data that needs to be inserted. For UPDATE and DELETE operations, the cursor identifies the rows that would be affected.

In PL/SQL, you can refer to the most recent implicit cursor as the SQL cursor, which always has the attributes like `%FOUND, %ISOPEN, %NOTFOUND, and %ROWCOUNT`.

| Attribute | Description |
|---|---|
| %FOUND | Returns TRUE if an INSERT, UPDATE, or DELETE statement affected one or more rows or a SELECT INTO statement returned one or more rows. Otherwise, it returns FALSE. |
| %NOTFOUND | The logical opposite of %FOUND. It returns TRUE if an INSERT, UPDATE, or DELETE statement affected no rows, or a SELECT INTO statement returned no rows. Otherwise, it returns FALSE. |
| %ISOPEN | Always returns FALSE for implicit cursors, because Oracle closes the SQL cursor automatically after executing its associated SQL statement. |
| %ROWCOUNT | Returns the number of rows affected by an INSERT, UPDATE, or DELETE statement, or returned by a SELECT INTO statement. |

**Example: Implicit Cursor**

```
    DECLARE
   total_rows number(2);
BEGIN
   UPDATE emp
   SET salary = salary + 500;
   IF sql%notfound THEN
      dbms_output.put_line('no customers selected');
   ELSIF sql%found THEN
      total_rows := sql%rowcount;
      dbms_output.put_line( total_rows || ' customers selected ');
   END IF;
END;
/
```

### 8.1.2 Explicit Cursors

Explicit cursors are programmer defined cursors for gaining more control over the context area. An explicit cursor should be defined in the declaration section of the PL/SQL Block. It is created on a SELECT Statement which returns one or more rows.

```
  CURSOR cursor_name IS select_statement;
```

**4 Steps for Cursors:**

1. Declaring the cursor for initializing in the memory
2. Opening the cursor for allocating memory
3. Fetching the cursor for retrieving data
4. Closing the cursor to release allocated memory

*Similar to typical file operation.*
**Example:**

```
 DECLARE
   c_id customers.id%type;
   c_name customers.name%type;
   c_addr customers.address%type;
   CURSOR c_customers is
      SELECT id, name, address FROM customers;
BEGIN
   OPEN c_customers;
   LOOP
      FETCH c_customers into c_id, c_name, c_addr;
      EXIT WHEN c_customers%notfound;
      dbms_output.put_line(c_id || ' ' || c_name || ' ' || c_addr);
   END LOOP;
   CLOSE c_customers;
END;
/
```

**CURSOR FOR Loop:** The cursor FOR loop is an elegant and natural extension of the numeric FOR loop in PL/SQL. With a numeric FOR loop, the body of the loop executes once for every integer value between the low and high values specified in the range. With a cursor FOR loop, the body of the loop is executed for each row returned by the query.
**Syntax**

```
  FOR record_index in cursor_name
LOOP
   {...statements...}
END LOOP;
```

**Example**

```
 CREATE OR REPLACE Function TotalIncome
   ( name_in IN varchar2 )
   RETURN varchar2
IS
   total_val number(6);

   cursor c1 is
     SELECT monthly_income
     FROM employees
     WHERE name = name_in;

BEGIN

   total_val := 0;

   FOR employee_rec in c1
   LOOP
      total_val := total_val + employee_rec.monthly_income;
   END LOOP;

   RETURN total_val;

END;
```

#### 8.1.2.1 Variables in Explicit Cursor Queries

An explicit cursor query can reference any variable in its scope. When you open an explicit cursor, PL/SQL evaluates any variables in the query and uses those values when identifying the result set. *Changing the values of the variables later does not change the result set.*

In the following Example, the explicit cursor query references the variable factor. When the cursor opens, factor has the value 2. Therefore, sal_multiple is always 2 times sal, despite that factor is incremented after every fetch.

```
 DECLARE
  sal           employees.salary%TYPE;
  sal_multiple  employees.salary%TYPE;
  factor        INTEGER := 2;

  CURSOR c1 IS
    SELECT salary, salary*factor FROM employees
    WHERE job_id LIKE 'AD_%';

BEGIN
  OPEN c1;  -- PL/SQL evaluates factor
```

```
  LOOP
    FETCH c1 INTO sal, sal_multiple;
    EXIT WHEN c1%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('factor = ' || factor);
    DBMS_OUTPUT.PUT_LINE('sal          = ' || sal);
    DBMS_OUTPUT.PUT_LINE('sal_multiple = ' || sal_multiple);
    factor := factor + 1;  -- Does not affect sal_multiple
  END LOOP;

  CLOSE c1;
END;
/


Result:

factor = 2
sal          = 4451
sal_multiple = 8902
factor = 3
sal          = 26460
sal_multiple = 52920
factor = 4
sal          = 18742.5
sal_multiple = 37485
factor = 5
sal          = 18742.5
sal_multiple = 37485
```

#### 8.1.2.2 Explicit Cursors that Accept Parameters

You can create an explicit cursor that has formal parameters, and then pass different actual parameters to the cursor each time you open it. In the cursor query, you can use a formal cursor parameter anywhere that you can use a constant. Outside the cursor query, you cannot reference formal cursor parameters.

Following Example creates an explicit cursor whose two formal parameters represent a job and its maximum salary. When opened with a specified job and maximum salary, the cursor query selects the employees with that job who are overpaid (for each such employee, the query selects the first and last name and amount overpaid). Next, the example creates a procedure that prints the cursor query result set. Finally, the example opens the cursor with one set of actual parameters, prints the result set, closes the cursor, opens the cursor with different actual parameters, prints the result set, and closes the cursor.

```
DROP TABLE EMPLOYEES;
```

```
CREATE TABLE EMPLOYEES
(ID NUMBER PRIMARY KEY,
 FIRST_NAME VARCHAR2(10),
 LAST_NAME VARCHAR2(10),
 JOB_ID VARCHAR2(10),
 SALARY NUMBER,
 CONSTRAINTS CHK_JOBID CHECK (JOB_ID IN ('ADMIN','MANAGER','FACULTY'))
);

---now insert some data ---

INSERT INTO EMPLOYEES VALUES(1,'A','B','ADMIN',30000);
INSERT INTO EMPLOYEES VALUES(2,'C','D','ADMIN',25000);
INSERT INTO EMPLOYEES VALUES(3,'E','F','ADMIN',33000);


INSERT INTO EMPLOYEES VALUES(4,'G','H','FACULTY',80000);
INSERT INTO EMPLOYEES VALUES(5,'I','J','FACULTY',120000);


INSERT INTO EMPLOYEES VALUES(6,'K','L','MANAGER',52000);
INSERT INTO EMPLOYEES VALUES(7,'M','N','MANAGER',50000);

COMMIT;


 ---Now create an annonymous block to create a parameterized cursor

 DECLARE
  CURSOR c (job VARCHAR2, max_sal NUMBER) IS
    SELECT last_name, first_name, (salary - max_sal) overpayment
    FROM employees
    WHERE job_id = job
    AND salary > max_sal
    ORDER BY salary;

  PROCEDURE print_overpaid IS
    last_name_   employees.last_name%TYPE;
    first_name_  employees.first_name%TYPE;
    overpayment_      employees.salary%TYPE;
  BEGIN
    LOOP
      FETCH c INTO last_name_, first_name_, overpayment_;
```

```
      EXIT WHEN c%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE(last_name_ || ', ' || first_name_ ||
        ' (by ' || overpayment_ || ')');
    END LOOP;
  END print_overpaid;

BEGIN
  DBMS_OUTPUT.PUT_LINE('----------------------');
  DBMS_OUTPUT.PUT_LINE('Overpaid ADMIN:');
  DBMS_OUTPUT.PUT_LINE('----------------------');
  OPEN c('ADMIN', 25000); ---call it for ADMIN people only
  print_overpaid;
  CLOSE c;

  DBMS_OUTPUT.PUT_LINE('------------------------------');
  DBMS_OUTPUT.PUT_LINE('Overpaid Sales FACULTY:');
  DBMS_OUTPUT.PUT_LINE('------------------------------');

  OPEN c('FACULTY', 100000); ---call it for FACULTY people only
  print_overpaid;
  CLOSE c;

DBMS_OUTPUT.PUT_LINE('------------------------------');
DBMS_OUTPUT.PUT_LINE('Overpaid Sales MANAGER:');
DBMS_OUTPUT.PUT_LINE('------------------------------');


OPEN c('MANAGER', 40000);---call it for MANAGER people only
print_overpaid;
CLOSE c;

END;
/


Result:


----------------------
Overpaid ADMIN:
----------------------
B, A (by 5000)
F, E (by 8000)
------------------------------
```

```
Overpaid Sales FACULTY:
------------------------------
J, I (by 20000)
------------------------------
Overpaid Sales MANAGER:
------------------------------
N, M (by 10000)
L, K (by 12000)

PL/SQL procedure successfully completed.
```

#### 8.1.2.3 SELECT...FOR UPDATE

When you issue a SELECT statement against the database to query some records, no locks are placed on the selected rows. In general, this is a wonderful feature because the number of records locked at any given time is kept to the absolute minimum: only those records that have been changed but not yet committed are locked. Even then, others are able to read those records as they appeared before the change (the before image of the data).

There are times, however, when you will want to lock a set of records even before you change them in your program. Oracle offers the FOR UPDATE clause of the SELECT statement to perform this locking.

When you issue a SELECT...FOR UPDATE statement, the database automatically obtains row-level locks on all the rows identified by the SELECT statement, holding the records for your changes only as you move through the rows retrieved by the cursor. Its as if youve issued an UPDATE statement against the rows, but you haventyouve merely SELECTed them. No one else will be able to change any of these records until you perform a ROLLBACK or a COMMITbut other sessions can still read the data.

**An Example:** Open one session (session 1) and do the followings:

```
 DECLARE
CURSOR c  IS
SELECT last_name, first_name, salary overpayment
FROM employees
WHERE job_id ='ADMIN'
FOR UPDATE;

BEGIN
OPEN c;

close c;
END;
```

Now open another session (session 2) (same user name) and try to update the same set of data as selected by the previous session (cursor);

```
update employees
set last_name='chName'
where ID between 1 and 3;
```

**Now Observe:** If you are at session one, it will be executed smoothly. In the session 2 you will find it is waiting for a transaction commands (commit or rollback) from other session (i.e. session 1). Whenever you issue commit , session 2 will be successful.

**Note:** It is possible to lock a specific field also instead of all fields in the cursor declaration. Like this:

```
CURSOR c  IS
SELECT last_name, first_name, salary overpayment
FROM employees
WHERE job_id ='ADMIN'
FOR UPDATE OF salary;
```

Here *only salary* will be locked in the selected records by the cursor.

**Releasing Locks with COMMIT:** As soon as a cursor with a FOR UPDATE clause is OPENed, all rows identified in the result set of the cursor are locked and remain locked until your session ends or your code explicitly issues either a COMMIT or a ROLLBACK. When either of these occurs, the locks on the rows are released. As a result, you cannot execute another FETCH against a FOR UPDATE cursor after a COMMIT or ROLLBACK. You will have lost your position in the cursor.

Consider the following program:

```
DECLARE
/* All the jobs in the fall to prepare for the winter */
CURSOR fall_jobs_cur
IS
SELECT task, expected_hours, tools_required, do_it_yourself_flag
FROM winterize
WHERE year = TO_NUMBER (TO_CHAR (SYSDATE, 'YYYY'))
AND completed_flag = 'NOTYET' FOR UPDATE;
BEGIN
/* For each job fetched by the cursor... */
FOR job_rec IN fall_jobs_cur
LOOP
IF job_rec.do_it_yourself_flag = 'YOUCANDOIT'
THEN
/*
|| I have found my next job. Assign it to myself (like someone
|| else is going to do it!) and then commit the changes.
*/
UPDATE winterize SET responsible = 'STEVEN'
WHERE task = job_rec.task
AND year = TO_NUMBER (TO_CHAR (SYSDATE, 'YYYY'));
COMMIT;
END IF;
END LOOP;
END;
```

When it tries to FETCH the second record, the program raises the following exception (as a COMMIT has been issued inside a locked session):

```
ORA-01002: fetch out of sequence
```