E

REF Cursors

Cursor Variables

- Cursor variables are like C or Pascal pointers, which hold the memory location (address) of an item instead of the item itself.
- In PL/SQL, a pointer is declared as `REF X`, where `REF` is short for `REFERENCE` and `X` stands for a class of objects.
- A cursor variable has the data type `REF CURSOR`.
- A cursor is static, but a cursor variable is dynamic.
- Cursor variables give you more flexibility.

ORACLE

Cursor variables are like C or Pascal pointers, which hold the memory location (address) of an item instead of the item itself. Thus, declaring a cursor variable creates a pointer, not an item. In PL/SQL, a pointer has the data type `REF X`, where `REF` is short for `REFERENCE` and `X` stands for a class of objects. A cursor variable has the `REF CURSOR` data type.

Like a cursor, a cursor variable points to the current row in the result set of a multirow query. However, cursors differ from cursor variables the way constants differ from variables. A cursor is static, but a cursor variable is dynamic because it is not tied to a specific query. You can open a cursor variable for any type-compatible query. This gives you more flexibility.

Cursor variables are available to every PL/SQL client. For example, you can declare a cursor variable in a PL/SQL host environment such as an OCI or Pro*C program, and then pass it as an input host variable (bind variable) to PL/SQL. Moreover, application development tools such as Oracle Forms and Oracle Reports, which have a PL/SQL engine, can use cursor variables entirely on the client side. The Oracle Server also has a PL/SQL engine. You can pass cursor variables back and forth between an application and server through remote procedure calls (RPCs).

Using Cursor Variables

- You can use cursor variables to pass query result sets between PL/SQL stored subprograms and various clients.
- PL/SQL can share a pointer to the query work area in which the result set is stored.
- You can pass the value of a cursor variable freely from one scope to another.
- You can reduce network traffic by having a PL/SQL block open (or close) several host cursor variables in a single roundtrip.

You use cursor variables to pass query result sets between PL/SQL stored subprograms and various clients. Neither PL/SQL nor any of its clients owns a result set; they simply share a pointer to the query work area in which the result set is stored. For example, an OCI client, an Oracle Forms application, and the Oracle Server can all refer to the same work area.

A query work area remains accessible as long as any cursor variable points to it. Therefore, you can pass the value of a cursor variable freely from one scope to another. For example, if you pass a host cursor variable to a PL/SQL block that is embedded in a Pro*C program, the work area to which the cursor variable points remains accessible after the block completes.

If you have a PL/SQL engine on the client side, calls from the client to the server impose no restrictions. For example, you can declare a cursor variable on the client side, open and fetch from it on the server side, and then continue to fetch from it back on the client side. Also, you can reduce network traffic by having a PL/SQL block open (or close) several host cursor variables in a single roundtrip.

A cursor variable holds a reference to the cursor work area in the Program Global Area (PGA) instead of addressing it with a static name. Because you address this area by a reference, you gain the flexibility of a variable.

Defining REF CURSOR Types

Define a REF CURSOR type:

```
Define a REF CURSOR type
TYPE ref_type_name IS REF CURSOR [RETURN return_type];
```

Declare a cursor variable of that type:

```
ref_cv ref_type_name;
```

Example:

```
DECLARE
TYPE DeptCurTyp IS REF CURSOR RETURN
departments%ROWTYPE;
dept_cv DeptCurTyp;
```


To define a REF CURSOR, you perform two steps. First, you define a REF CURSOR type, and then you declare cursor variables of that type. You can define REF CURSOR types in any PL/SQL block, subprogram, or package using the following syntax:

```
TYPE ref_type_name IS REF CURSOR [RETURN return_type];
```

where:

ref_type_name	Is a type specifier used in subsequent declarations of cursor variables
return_type	Represents a record or a row in a database table

In this example, you specify a return type that represents a row in the database table DEPARTMENT.

REF CURSOR types can be strong (restrictive) or weak (nonrestrictive). As the next example shows, a strong REF CURSOR type definition specifies a return type, but a weak definition does not:

```
DECLARE
   TYPE EmpCurTyp IS REF CURSOR RETURN employees%ROWTYPE;  -- strong
   TYPE GenericCurTyp IS REF CURSOR;  -- weak
```

Strong REF CURSOR types are less error prone because the PL/SQL compiler lets you associate a strongly typed cursor variable only with type-compatible queries. However, weak REF CURSOR types are more flexible because the compiler lets you associate a weakly typed cursor variable with any query.

Declaring Cursor Variables

After you define a REF CURSOR type, you can declare cursor variables of that type in any PL/SQL block or subprogram. In the following example, you declare the cursor variable DEPT_CV:

```
DECLARE
   TYPE DeptCurTyp IS REF CURSOR RETURN departments%ROWTYPE;
   dept_cv DeptCurTyp;  -- declare cursor variable
```

Note: You cannot declare cursor variables in a package. Unlike packaged variables, cursor variables do not have persistent states. Remember, declaring a cursor variable creates a pointer, not an item. Cursor variables cannot be saved in the database; they follow the usual scoping and instantiation rules.

In the RETURN clause of a REF CURSOR type definition, you can use %ROWTYPE to specify a record type that represents a row returned by a strongly (not weakly) typed cursor variable, as follows:

```
DECLARE
   TYPE TmpCurTyp IS REF CURSOR RETURN employees%ROWTYPE;
   tmp_cv TmpCurTyp;  -- declare cursor variable
   TYPE EmpCurTyp IS REF CURSOR RETURN tmp_cv%ROWTYPE;
   emp_cv EmpCurTyp;  -- declare cursor variable
```

Similarly, you can use %TYPE to provide the data type of a record variable, as the following example shows:

```
DECLARE
   dept_rec departments%ROWTYPE;  -- declare record variable
   TYPE DeptCurTyp IS REF CURSOR RETURN dept_rec%TYPE;
   dept_cv DeptCurTyp;  -- declare cursor variable
```

In the final example, you specify a user-defined RECORD type in the RETURN clause:

```
DECLARE
   TYPE EmpRecTyp IS RECORD (
      empno NUMBER(4),
      ename VARCHAR2(1O),
      sal   NUMBER(7,2));
   TYPE EmpCurTyp IS REF CURSOR RETURN EmpRecTyp;
   emp_cv EmpCurTyp;  -- declare cursor variable
```

Cursor Variables as Parameters

You can declare cursor variables as the formal parameters of functions and procedures. In the following example, you define the `REF CURSOR` type `EmpCurTyp`, and then declare a cursor variable of that type as the formal parameter of a procedure:

```
DECLARE
   TYPE EmpCurTyp IS REF CURSOR RETURN emp%ROWTYPE;
   PROCEDURE open_emp_cv (emp_cv IN OUT EmpCurTyp) IS ...
```

Using the OPEN-FOR, FETCH, and CLOSE Statements

- The OPEN-FOR statement associates a cursor variable with a multirow query, executes the query, identifies the result set, and positions the cursor to point to the first row of the result set.
- The FETCH statement returns a row from the result set of a multirow query, assigns the values of the select-list items to the corresponding variables or fields in the INTO clause, increments the count kept by %ROWCOUNT, and advances the cursor to the next row.
- The CLOSE statement disables a cursor variable.

ORACLE

You use three statements to process a dynamic multirow query: OPEN-FOR, FETCH, and CLOSE. First, you "open" a cursor variable "for" a multirow query. Then you "fetch" rows from the result set one at a time. When all the rows are processed, you "close" the cursor variable.

Opening the Cursor Variable

The OPEN-FOR statement associates a cursor variable with a multirow query, executes the query, identifies the result set, positions the cursor to point to the first row of the results set, and then sets the rows-processed count kept by %ROWCOUNT to zero. Unlike the static form of OPEN-FOR, the dynamic form has an optional USING clause. At run time, bind arguments in the USING clause replace corresponding placeholders in the dynamic SELECT statement. The syntax is:

```
OPEN {cursor_variable | :host_cursor_variable} FOR
dynamic_string
            [USING bind_argument[, bind_argument]...];
```

where CURSOR_VARIABLE is a weakly typed cursor variable (one without a return type), HOST_CURSOR_VARIABLE is a cursor variable declared in a PL/SQL host environment such as an OCI program, and dynamic_string is a string expression that represents a multirow query.

In the following example, the syntax declares a cursor variable, and then associates it with a dynamic SELECT statement that returns rows from the employees table:

```
DECLARE
 TYPE EmpCurTyp IS REF CURSOR;  -- define weak REF CURSOR      type
 emp_cv   EmpCurTyp;  -- declare cursor variable
 my_ename VARCHAR2(15);
 my_sal   NUMBER := 1000;
BEGIN
 OPEN emp_cv FOR  -- open cursor variable
    'SELECT last_name, salary FROM employees WHERE salary >
         :s'
     USING my_sal;
 ...
END;
```

Any bind arguments in the query are evaluated only when the cursor variable is opened. Thus, to fetch rows from the cursor using different bind values, you must reopen the cursor variable with the bind arguments set to their new values each time.

Fetching from the Cursor Variable

The FETCH statement returns a row from the result set of a multirow query, assigns the values of the select-list items to the corresponding variables or fields in the INTO clause, increments the count kept by %ROWCOUNT, and advances the cursor to the next row. Use the following syntax:

```
FETCH {cursor_variable | :host_cursor_variable}
 INTO {define_variable[, define_variable]... | record};
```

Continuing the example, fetch rows from the cursor variable emp_cv into the define variables MY_ENAME and MY_SAL:

```
LOOP
 FETCH emp_cv INTO my_ename, my_sal;  -- fetch next row
 EXIT WHEN emp_cv%NOTFOUND;  -- exit loop when last row is
   fetched
 -- process row
END LOOP;
```

For each column value returned by the query associated with the cursor variable, there must be a corresponding, type-compatible variable or field in the INTO clause. You can use a different INTO clause on separate fetches with the same cursor variable. Each fetch retrieves another row from the same result set. If you try to fetch from a closed or never-opened cursor variable, PL/SQL raises the predefined exception INVALID_CURSOR.

Closing the Cursor Variable

The CLOSE statement disables a cursor variable. After that, the associated result set is undefined. Use the following syntax:

```
CLOSE {cursor_variable | :host_cursor_variable};
```

In this example, when the last row is processed, close the emp_cv cursor variable:

```
LOOP
 FETCH emp_cv INTO my_ename, my_sal;
 EXIT WHEN emp_cv%NOTFOUND;
 -- process row
END LOOP;
CLOSE emp_cv;  -- close cursor variable
```

If you try to close an already-closed or never-opened cursor variable, PL/SQL raises INVALID_CURSOR.

Example of Fetching

```
DECLARE
   TYPE EmpCurTyp IS REF CURSOR;
   emp_cv   EmpCurTyp;
   emp_rec  employees%ROWTYPE;
   sql_stmt VARCHAR2(200);
   my_job   VARCHAR2(10) := 'ST_CLERK';
BEGIN
   sql_stmt := 'SELECT * FROM employees
                WHERE job_id = :j';
   OPEN emp_cv FOR sql_stmt USING my_job;
   LOOP
      FETCH emp_cv INTO emp_rec;
      EXIT WHEN emp_cv%NOTFOUND;
      -- process record
   END LOOP;
   CLOSE emp_cv;
END;
/
```


The example in the slide shows that you can fetch rows from the result set of a dynamic multirow query into a record. You must first define a `REF CURSOR` type, `EmpCurTyp`. You then define a cursor variable `emp_cv`, of the type `EmpcurTyp`. In the executable section of the PL/SQL block, the `OPEN-FOR` statement associates the cursor variable `emp_cv` with the multirow query, `sql_stmt`. The `FETCH` statement returns a row from the result set of a multirow query and assigns the values of the select-list items to `EMP_REC` in the `INTO` clause. When the last row is processed, close the `emp_cv` cursor variable.