

2

Design Considerations

ORACLE

Copyright © 2004, Oracle. All rights reserved.

Objectives

After completing this lesson, you should be able to do the following:

- **Identify guidelines for cursor design**
- **Use cursor variables**
- **Create subtypes based on existing types for an application**

ORACLE

Copyright © 2004, Oracle. All rights reserved.

Objectives

This lesson discusses several concepts that apply to the designing of PL/SQL program units. This lesson explains how to:

- Design and use cursor variables
- Describe the predefined data types
- Create subtypes based on existing data types for an application

Guidelines for Cursor Design

Fetch into a record when fetching from a cursor.

```
DECLARE
  CURSOR cur_cust IS
    SELECT customer_id, cust_last_name, cust_email
    FROM customers
    WHERE credit_limit = 1200;
  v_cust_record    cur_cust%ROWTYPE;
BEGIN
  OPEN cur_cust;
  LOOP
    FETCH cur_cust INTO v_cust_record;
  ...
```

ORACLE

Copyright © 2004, Oracle. All rights reserved.

Guidelines for Cursor Design

When fetching from a cursor, fetch into a record. This way you do not need to declare individual variables, and you reference only the values you want to use. Additionally, you can automatically use the structure of the SELECT column list.

Guidelines for Cursor Design

Create cursors with parameters.

```
CREATE OR REPLACE PROCEDURE cust_pack
(p_crd_limit_in NUMBER, p_acct_mgr_in NUMBER)
IS
  v_credit_limit NUMBER := 1500;
  CURSOR cur_cust
    (p_crd_limit NUMBER, p_acct_mgr NUMBER)
  IS
    SELECT customer_id, cust_last_name, cust_email
    FROM customers
    WHERE credit_limit = p_crd_limit
    AND   account_mgr_id = p_acct_mgr;
  cust_record      cur_cust%ROWTYPE;
BEGIN
  OPEN cur_cust(p_crd_limit_in, p_acct_mgr_in);
  ...
  CLOSE cur_cust;
  ...
  OPEN cur_cust(v_credit_limit, 145);
  ...
END;
```

The diagram illustrates how parameters are passed to a cursor. It shows a cursor definition with two parameters, `p_crd_limit` and `p_acct_mgr`. Below the definition, two `OPEN` statements are shown. The first `OPEN` statement passes `p_crd_limit_in` and `p_acct_mgr_in` as arguments. The second `OPEN` statement passes `v_credit_limit` and `145` as arguments. Arrows point from the arguments in the `OPEN` statements to the parameter names in the cursor definition, indicating the mapping of values to parameters.

ORACLE

Copyright © 2004, Oracle. All rights reserved.

Guidelines for Cursor Design (continued)

Whenever you have a need to use a cursor in more than one place with different values for the WHERE clause, create parameters for your cursor. Parameters increase the flexibility and reusability of cursors, because you can pass different values to the WHERE clause when you open a cursor, rather than hard-code a value for the WHERE clause.

Additionally, parameters help you avoid scoping problems, because the result set for the cursor is not tied to a specific variable in a program. You can define a cursor at a higher level and use it in any subblock with variables defined in the local block.

Guidelines for Cursor Design

Reference implicit cursor attributes immediately after the SQL statement executes.

```
BEGIN
  UPDATE customers
    SET   credit_limit = p_credit_limit
    WHERE customer_id = p_cust_id;
  → get_avg_order(p_cust_id); -- procedure call
  IF SQL%NOTFOUND THEN
    ...
```

ORACLE

Copyright © 2004, Oracle. All rights reserved.

Guidelines for Cursor Design (continued)

If you are using an implicit cursor and reference a SQL cursor attribute, make sure you reference it immediately after a SQL statement is executed. This is because SQL cursor attributes are set on the result of the most recently executed SQL statement. The SQL statement can be executed in another program. Referencing a SQL cursor attribute immediately after a SQL statement executes ensures that you are dealing with the result of the correct SQL statement.

In the example in the slide, you cannot rely on the value of SQL%NOTFOUND for the UPDATE statement, because it is likely to be overwritten by the value of another SQL statement in the `get_avg_order` procedure. To ensure accuracy, the cursor attribute function SQL%NOTFOUND needs to be called immediately after the DML statement:

```
DECLARE
  v_flag BOOLEAN;
BEGIN
  UPDATE customers
    SET   credit_limit = p_credit_limit
    WHERE customer_id = p_cust_id;
  v_flag := SQL%NOTFOUND
  get_avg_order(p_cust_id); -- procedure call
  IF v_flag THEN
    ...
```

Guidelines for Cursor Design

Simplify coding with cursor FOR loops.

```
CREATE OR REPLACE PROCEDURE cust_pack
(p_crd_limit_in NUMBER, p_acct_mgr_in NUMBER)
IS
  v_credit_limit NUMBER := 1500;
  CURSOR cur_cust
    (p_crd_limit NUMBER, p_acct_mgr NUMBER)
  IS
    SELECT customer_id, cust_last_name, cust_email
    FROM customers
    WHERE credit_limit = p_crd_limit
    AND   account_mgr_id = p_acct_mgr;
  cust_record      cur_cust%ROWTYPE;
BEGIN
  FOR cust_record IN cur_cust
    (p_crd_limit_in, p_acct_mgr_in)
  LOOP
    -- implicit open and fetch
    ...
  END LOOP;
  -- implicit close
  ...
END;
```

ORACLE

Copyright © 2004, Oracle. All rights reserved.

Guidelines for Cursor Design (continued)

Whenever possible, use cursor FOR loops that simplify coding. Cursor FOR loops reduce the volume of code you need to write to fetch data from a cursor and also reduce the chances of introducing loop errors in your code.

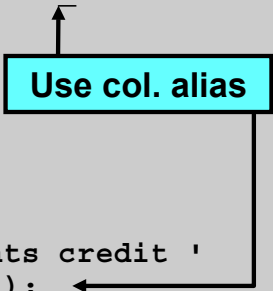
A cursor FOR loop automatically handles the open, fetch, and close operations, as well as, defines a record type that matches the cursor definition. After it processes the last row the cursor is closed automatically. If you do not use a CURSOR FOR loop, forgetting to close your cursor results in increased memory usage.

Guidelines for Cursor Design

- Close a cursor when it is no longer needed.
- Use column aliases in cursors for calculated columns fetched into records declared with %ROWTYPE.

```
CREATE OR REPLACE PROCEDURE cust_list
IS
  CURSOR cur_cust IS
    SELECT customer_id, cust_last_name, credit_limit*1.1
    FROM customers;
  cust_record cur_cust%ROWTYPE;
BEGIN
  OPEN cur_cust;
  LOOP
    FETCH cur_cust INTO cust_record;
    DBMS_OUTPUT.PUT_LINE('Customer ' ||
      cust_record.cust_last_name || ' wants credit '
      || cust_record.(credit_limit * 1.1));
    EXIT WHEN cur_cust%NOTFOUND;
  END LOOP;
  ...

```



ORACLE

Copyright © 2004, Oracle. All rights reserved.

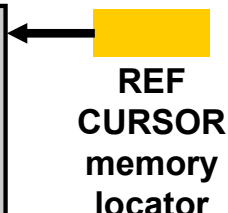
Guidelines for Cursor Design (continued)

- If you do not need a cursor any longer, close it explicitly. If your cursor is in a package, its scope is not limited to any particular PL/SQL block. The cursor remains open until you explicitly close it. An open cursor takes up memory space and continues to maintain row-level locks, if created with the FOR UPDATE clause, until a commit or rollback. Closing the cursor releases memory. Ending the transaction by committing or rolling back releases the locks. Along with a FOR UPDATE clause you can also use a WHERE CURRENT OF clause with the DML statements inside the FOR loop. This automatically performs a DML transaction for the current row in the cursor's result set, thereby improving performance. **Note:** It is a good programming practice to explicitly close your cursors. Leaving cursors open can generate an exception because the number of cursors allowed to remain open within a session is limited.
- Make sure that you use column aliases in your cursor for calculated columns that you fetch into a record declared with a %ROWTYPE declaration. You also need column aliases if you want to reference the calculated column in your program. The code in the slide does not compile successfully because it lacks a column alias for the calculation `credit_limit*1.1`. After you give it an alias, use the same alias later in the code to make a reference to the calculation.

Cursor Variables

Memory

1	Southlake, Texas	1400
2	San Francisco	1500
3	New Jersey	1600
4	Seattle, Washington	1700
5	Toronto	1800



A yellow rectangular box labeled "REF CURSOR memory locator" has an arrow pointing to the right side of the table.

ORACLE

Copyright © 2004, Oracle. All rights reserved.

Cursor Variables: Overview

Like a cursor, a cursor variable points to the current row in the result set of a multirow query. Cursor variables, however, are like C pointers: they hold the memory location of an item instead of the item itself. In this way, cursor variables differ from cursors the way constants differ from variables. A cursor is static, a cursor variable is dynamic. In PL/SQL, a cursor variable has a **REF CURSOR** data type, where REF stands for reference, and CURSOR stands for the class of the object.

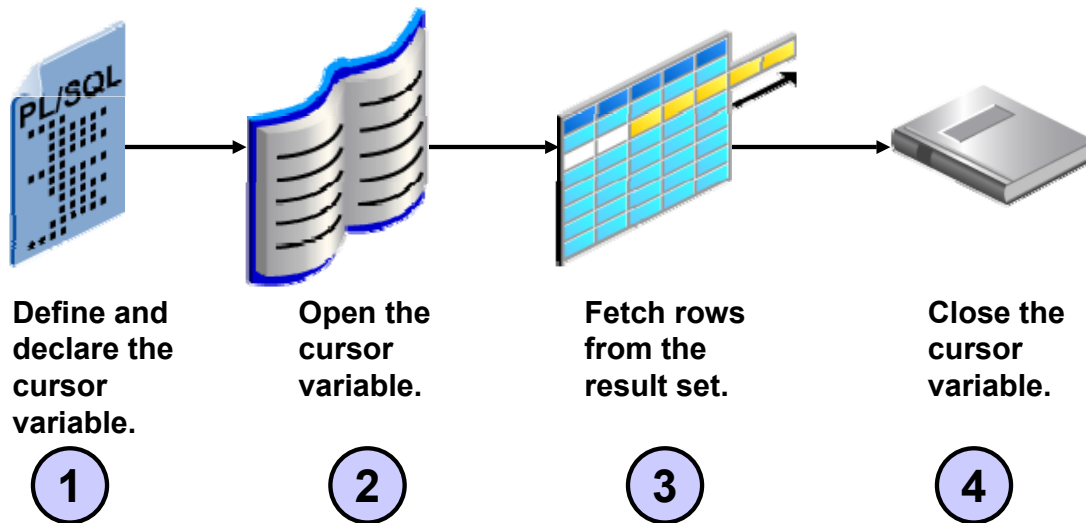
Using Cursor Variables

To execute a multirow query, the Oracle server opens a work area called a “cursor” to store processing information. To access the information, you either explicitly name the work area, or you use a cursor variable that points to the work area. Whereas a cursor always refers to the same work area, a cursor variable can refer to different work areas. Therefore, cursors and cursor variables are not interoperable.

An explicit cursor is static and is associated with one SQL statement. A cursor variable can be associated with different statements at run time.

Primarily you use a cursor variable to pass a pointer to query result sets between PL/SQL stored subprograms and various clients such as a Developer Forms application. None of them owns the result set. They simply share a pointer to the query work area that stores the result set.

Using a Cursor Variable



ORACLE

Copyright © 2004, Oracle. All rights reserved.

Working with Cursor Variables

There are four steps for handling a cursor variable. The next few sections contain detailed information about each step.

Strong Versus Weak Cursors

- **Strong cursor:**
 - Is restrictive
 - Specifies a RETURN type
 - Associates with type-compatible queries only
 - Is less error prone
- **Weak cursor:**
 - Is nonrestrictive
 - Associates with any query
 - Is very flexible

ORACLE

Copyright © 2004, Oracle. All rights reserved.

Strong Versus Weak Cursor Variables

REF CURSOR types can be strong (restrictive) or weak (nonrestrictive). A strong REF CURSOR type definition specifies a return type, a weak definition does not. PL/SQL enables you to associate a strong type with type-compatible queries only, whereas a weak type can be associated with any query. This makes strong REF CURSOR types less error prone, but weak REF CURSOR types more flexible.

In the following example, the first definition is strong, whereas the second is said to be weak:

```
DECLARE
  TYPE rt_cust IS REF CURSOR RETURN customers%ROWTYPE;
  TYPE rt_general_purpose IS REF CURSOR;
  ...
```

Step 1: Defining a REF CURSOR Type

Define a REF CURSOR type:

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

- *ref_type_name* is a type specifier in subsequent declarations.
- *return_type* represents a record type.
- *return_type* indicates a strong cursor.

```
DECLARE
  TYPE rt_cust IS REF CURSOR
    RETURN customers%ROWTYPE;
  ...
```

ORACLE

Copyright © 2004, Oracle. All rights reserved.

Step 1: Defining a Cursor Variable

To create a cursor variable, you first need to define a REF CURSOR type and then declare a variable of that type.

Defining the REF CURSOR type:

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

where: *ref_type_name* a type specified in subsequent declarations
 return_type represents a row in a database table

The REF keyword indicates that the new type is to be a pointer to the defined type. The *return_type* is a record type indicating the types of the select list that are eventually returned by the cursor variable. The return type must be a record type.

Example

```
DECLARE
  TYPE rt_cust IS REF CURSOR RETURN customers%ROWTYPE;
  ...
```

Step 1: Declaring a Cursor Variable

Declare a cursor variable of a cursor type:

```
CURSOR_VARIABLE_NAME      REF_TYPE_NAME
```

- *cursor_variable_name* is the name of the cursor variable.
- *ref_type_name* is the name of a REF CURSOR type.

```
DECLARE
  TYPE rt_cust IS REF CURSOR
    RETURN customers%ROWTYPE;
  cv_cust rt_cust;
```

ORACLE

Copyright © 2004, Oracle. All rights reserved.

Declaring a Cursor Variable

After the cursor type is defined, declare a cursor variable of that type.

```
cursor_variable_name      ref_type_name;
```

where: *cursor_variable_name* is the name of the cursor variable
 ref_type_name is the name of the REF CURSOR type

Cursor variables follow the same scoping and instantiation rules as all other PL/SQL variables.

In the following example, you declare the cursor variable *cv_cust*.

Step 1:

```
DECLARE
  TYPE ct_cust IS REF CURSOR RETURN customers%ROWTYPE;
  cv_cust rt_cust;
```

Step 1: Declaring a REF CURSOR Return Type

Options:

- **Use %TYPE and %ROWTYPE.**
- **Specify a user-defined record in the RETURN clause.**
- **Declare the cursor variable as the formal parameter of a stored procedure or function.**

ORACLE

Copyright © 2004, Oracle. All rights reserved.

Step 1: Declaring a REF CURSOR Return Type

The following are some more examples of cursor variable declarations:

- Use %TYPE and %ROWTYPE to provide the data type of a record variable:

```
DECLARE
    cust_rec customers%ROWTYPE; --a recd variable based on a row
    TYPE rt_cust IS REF CURSOR RETURN cust_rec%TYPE;
    cv_cust      rt_cust; --cursor variable
```

- Specify a user-defined record in the RETURN clause:

```
DECLARE
    TYPE cust_rec_typ IS RECORD
        (custno      NUMBER(4),
         custname    VARCHAR2(10),
         credit      NUMBER(7,2));
    TYPE rt_cust IS REF CURSOR RETURN cust_rec_typ;
    cv_cust      rt_cust;
```

- Declare a cursor variable as the formal parameter of a stored procedure or function:

```
DECLARE
    TYPE rt_cust IS REF CURSOR RETURN customers%ROWTYPE;
    PROCEDURE use_cust_cur_var(cv_cust IN OUT rt_cust)
    IS ...
```

Step 2: Opening a Cursor Variable

- **Associate a cursor variable with a multirow `SELECT` statement.**
- **Execute the query.**
- **Identify the result set:**

```
OPEN cursor_variable_name
  FOR select_statement
```

- *cursor_variable_name* is the name of the cursor variable.
- *select_statement* is the SQL `SELECT` statement.

ORACLE

Copyright © 2004, Oracle. All rights reserved.

Step 2: Opening a Cursor Variable

Other `OPEN-FOR` statements can open the same cursor variable for different queries. You do not need to close a cursor variable before reopening it. You must note that when you reopen a cursor variable for a different query, the previous query is lost.

In the following example, the packaged procedure declares a variable used to select one of several alternatives in an `IF THEN ELSE` statement. When called, the procedure opens the cursor variable for the chosen query.

```
CREATE OR REPLACE PACKAGE cust_data
IS
  TYPE rt_cust IS REF CURSOR RETURN customers%ROWTYPE;
  PROCEDURE open_cust_cur_var(cv_cust IN OUT rt_cust,
                              p_your_choice IN NUMBER);
END cust_data;
/
```

Step 2: Opening a Cursor Variable (continued)

```
CREATE OR REPLACE PACKAGE BODY cust_data
IS
    PROCEDURE open_cust_cur_var(cv_cust IN OUT rt_cust,
                                p_your_choice IN NUMBER)
    IS
    BEGIN
        IF p_your_choice = 1 THEN
            OPEN cv_cust FOR SELECT * FROM customers;
        ELSIF p_your_choice = 2 THEN
            OPEN cv_cust FOR SELECT * FROM customers
                                WHERE credit_limit > 3000;
        ELSIF p_your_choice = 3 THEN
            ...
        END IF;
    END open_cust_cur_var;
END cust_data;
/
```

Step 3: Fetching from a Cursor Variable

- Retrieve rows from the result set one at a time.

```
FETCH cursor_variable_name  
  INTO variable_name1  
        [,variable_name2,. . .]  
  | record_name;
```

- The return type of the cursor variable must be compatible with the variables named in the INTO clause of the FETCH statement.

ORACLE

Copyright © 2004, Oracle. All rights reserved.

Step 3: Fetching from a Cursor Variable

The FETCH statement retrieves rows from the result set one at a time. PL/SQL verifies that the return type of the cursor variable is compatible with the INTO clause of the FETCH statement. For each query column value returned, there must be a type-compatible variable in the INTO clause. Also, the number of query column values must equal the number of variables. In case of a mismatch in number or type, the error occurs at compile time for strongly typed cursor variables and at run time for weakly typed cursor variables.

Note: When you declare a cursor variable as the formal parameter of a subprogram that fetches from a cursor variable, you must specify the IN (or IN OUT) mode. If the subprogram also opens the cursor variable, you must specify the IN OUT mode.

Step 4: Closing a Cursor Variable

- **Disable a cursor variable.**
- **The result set is undefined.**

```
CLOSE cursor_variable_name ;
```

- **Accessing the cursor variable after it is closed raises the predefined exception `INVALID_CURSOR`.**

ORACLE

Copyright © 2004, Oracle. All rights reserved.

Step 4: Closing a Cursor Variable

The `CLOSE` statement disables a cursor variable. After that the result set is undefined. The syntax is:

```
CLOSE cursor_variable_name;
```

In the following example, the cursor is closed when the last row is processed.

```
...  
  LOOP  
    FETCH cv_cust INTO cust_rec;  
    EXIT WHEN cv_cust%NOTFOUND;  
    ...  
  END LOOP;  
  CLOSE cv_cust;  
  ...
```

Passing Cursor Variables as Arguments

You can pass query result sets among PL/SQL stored subprograms and various clients.



ORACLE

Copyright © 2004, Oracle. All rights reserved.

Passing Query Result Sets

Cursor variables are very useful for passing 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 that identifies the result set. For example, an Oracle Call Interface (OCI) client, or an Oracle Forms application, or the Oracle server can all refer to the same work area. This might be useful in Oracle Forms, for instance, when you want to populate a multiblock form.

Example

Using SQL*Plus, define a host variable with a data type of REFCURSOR to hold the query results generated from a REF CURSOR in a stored subprogram. Use the SQL*Plus PRINT command to view the host variable results. Optionally, you can set the SQL*Plus command SET AUTOPRINT ON to display the query results automatically.

```
SQL> VARIABLE cv REFCURSOR
```

Next, create a subprogram that uses a REF CURSOR to pass the cursor variable data back to the SQL*Plus environment.

Passing Cursor Variables as Arguments

```
SQL> EXECUTE cust_data.get_cust(112, :cv)
```

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

```
SQL> print cv
```

CUSTOMER_ID	CUST_FIRST_NAME	CREDIT_LIMIT	CUST_EMAIL
112	Guillaume	200	Guillaume.Jackson@MOORHEN.COM

ORACLE

Copyright © 2004, Oracle. All rights reserved.

Passing Query Result Sets (continued)

```
CREATE OR REPLACE PACKAGE cust_data AS
TYPE typ_cust_rec IS RECORD
  (cust_id NUMBER(6), custname VARCHAR2(20),
   credit   NUMBER(9,2), cust_email VARCHAR2(30));
TYPE rt_cust IS REF CURSOR RETURN typ_cust_rec;
PROCEDURE get_cust
  (p_custid IN NUMBER, p_cv_cust IN OUT rt_cust);
END;
/
```

Passing Query Result Sets (continued)

```
CREATE OR REPLACE PACKAGE BODY cust_data AS
  PROCEDURE get_cust
    (p_custid IN NUMBER, p_cv_cust IN OUT rt_cust)
  IS
  BEGIN
    OPEN p_cv_cust FOR
    SELECT customer_id, cust_first_name, credit_limit, cust_email
      FROM customers
      WHERE customer_id = p_custid;
    -- CLOSE p_cv_cust
  END;
END;
/
```

Note that the `CLOSE p_cv_cust` statement is commented. This is done because if you close the REF cursor, it is not accessible from the host variable.

Rules for Cursor Variables

- **Cursor variables cannot be used with remote subprograms on another server.**
- **The query associated with a cursor variable in an OPEN-FOR statement should not be FOR UPDATE.**
- **You cannot use comparison operators to test cursor variables.**
- **Cursor variables cannot be assigned a null value.**
- **You cannot use REF CURSOR types in CREATE TABLE or VIEW statements.**
- **Cursors and cursor variables are not interoperable.**

ORACLE

Copyright © 2004, Oracle. All rights reserved.

Restrictions

- Remote subprograms on another server cannot accept the values of cursor variables. Therefore, you cannot use remote procedure calls (RPCs) to pass cursor variables from one server to another.
- If you pass a host cursor variable to PL/SQL, you cannot fetch from it on the server side unless you open it in the server on the same server call.
- The query associated with a cursor variable in an OPEN-FOR statement should not be FOR UPDATE.
- You cannot use comparison operators to test cursor variables for equality, inequality, or nullity.
- You cannot assign NULLs to a cursor variable.
- You cannot use REF CURSOR types to specify column types in a CREATE TABLE or CREATE VIEW statement. So, database columns cannot store the values of cursor variables.
- You cannot use a REF CURSOR type to specify the element type of a collection, which means that elements in an index-by table, nested table, or VARRAY cannot store the values of cursor variables.
- Cursors and cursor variables are not interoperable; that is, you cannot use one where the other is expected.

Comparing Cursor Variables with Static Cursors

Cursor variables have the following benefits:

- **Are dynamic and ensure more flexibility**
- **Are not tied to a single `SELECT` statement**
- **Hold the value of a pointer**
- **Can reduce network traffic**
- **Give access to query work area after a block completes**

ORACLE

Copyright © 2004, Oracle. All rights reserved.

Comparing Cursor Variables with Static Cursors

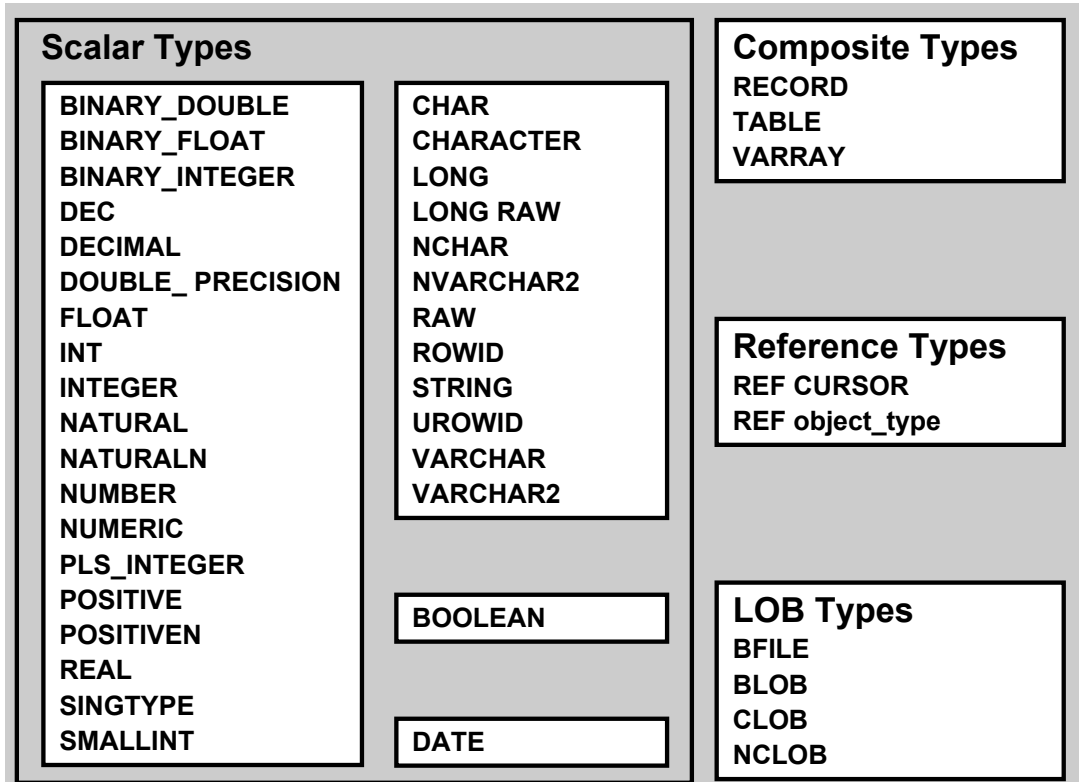
Cursor variables are dynamic and provide wider flexibility. Unlike static cursors, cursor variables are not tied to a single `SELECT` statement. In applications where the `SELECT` statement may differ depending on different situations, cursor variables can be opened for the different `SELECT` statement. Because cursor variables hold the value of a pointer, they can be easily passed between programs, no matter where the programs exist.

Cursor variables can reduce network traffic by grouping `OPEN FOR` statements and sending them across the network only once. For example, the following PL/SQL block opens two cursor variables in a single round trip:

```
/* anonymous PL/SQL block in host environment */
BEGIN
    OPEN :cv_cust FOR SELECT * FROM customers;
    OPEN :cv_orders FOR SELECT * FROM orders;
END;
```

This may be useful in Oracle Forms, for instance, when you want to populate a multiblock form. When you pass host cursor variables to a PL/SQL block for opening, the query work areas to which they point remain accessible after the block completes. That enables your OCI or Pro*C program to use these work areas for ordinary cursor operations.

Predefined Data Types



ORACLE

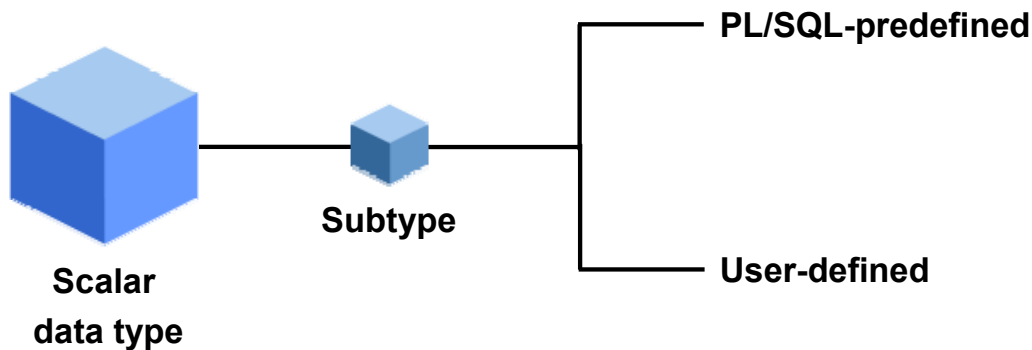
Copyright © 2004, Oracle. All rights reserved.

PL/SQL Data Types

Every constant, variable, and parameter has a data type, which specifies a storage format, constraints, and a valid range of values. PL/SQL provides a variety of predefined data types. For instance, you can choose from integer, floating point, character, Boolean, date, collection, reference, and LOB types. In addition, PL/SQL enables you to define your own subtypes.

Subtypes

A subtype is a subset of an existing data type that may place a constraint on its base type.



ORACLE

Copyright © 2004, Oracle. All rights reserved.

Definition of Subtypes

A subtype is a data type based on an existing data type. It does not define a new data type, instead it places a constraint on an existing data type. There are several predefined subsets specified in the standard package. `DECIMAL` and `INTEGER` are subtypes of `NUMBER`. `CHARACTER` is a subtype of `CHAR`.

Standard Subtypes

BINARY_INTEGER Subtypes	NUMBER Subtypes	VARCHAR2 Subtypes
NATURAL NATURALN POSITIVE POSITIVEN SIGNTYPE	DEC DECIMAL DOUBLE PRECISION FLOAT INTEGER INT NUMERIC REAL SMALLINT	STRING VARCHAR

Definition of Subtypes (continued)

With the `NATURAL` and `POSITIVE` subtypes, you can restrict an integer variable to non-negative and positive values, respectively. `NATURALN` and `POSITIVEN` prevent the assigning of nulls to an integer variable. You can use `SIGNTYPE` to restrict an integer variable to the values `-1`, `0`, and `1`, which is useful in programming tri-state logic.

A constrained subtype is a subset of the values normally specified by the data type on which the subtype is based. `POSITIVE` is a constrained subtype of `BINARY_INTEGER`.

An unconstrained subtype is not a subset of another data type; it is an alias to another data type. `FLOAT` is an unconstrained subtype of `NUMBER`.

Use the subtypes `DEC`, `DECIMAL`, and `NUMERIC` to declare fixed-point numbers with a maximum precision of 38 decimal digits.

Use the subtypes `DOUBLE PRECISION` and `FLOAT` to declare floating-point numbers with a maximum precision of 126 binary digits, which is roughly equivalent to 38 decimal digits. Or, use the subtype `REAL` to declare floating-point numbers with a maximum precision of 63 binary digits, which is roughly equivalent to 18 decimal digits.

Use the subtypes `INTEGER`, `INT`, and `SMALLINT` to declare integers with a maximum precision of 38 decimal digits.

You can create your own user-defined subtypes.

Note: You can use these subtypes for compatibility with ANSI/ISO and IBM types. Currently, `VARCHAR` is synonymous with `VARCHAR2`. However, in future releases of PL/SQL, to accommodate emerging SQL standards, `VARCHAR` may become a separate data type with different comparison semantics. It is a good idea to use `VARCHAR2` rather than `VARCHAR`.

Benefits of Subtypes

Subtypes:

- **Increase reliability**
- **Provide compatibility with ANSI/ISO and IBM types**
- **Promote reusability**
- **Improve readability**
 - **Clarity**
 - **Code self-documents**

ORACLE

Copyright © 2004, Oracle. All rights reserved.

Benefits

If your applications require a subset of an existing data type, you can create your own subtypes. By using subtypes, you can increase the reliability and improve the readability by indicating the intended use of constants and variables. Subtypes can increase reliability by detecting the out-of-range values.

With the predefined subtypes, you have compatibility with other data types from other programming languages.

Declaring Subtypes

- Subtypes are defined in the declarative section of any PL/SQL block.

```
SUBTYPE subtype_name IS base_type [(constraint)]
[NOT NULL];
```

- *subtype_name* is a type specifier used in subsequent declarations.
- *base_type* is any scalar or user-defined PL/SQL type.

ORACLE

Copyright © 2004, Oracle. All rights reserved.

Declaring Subtypes

Subtypes are defined in the declarative section of a PL/SQL block, subprogram, or package. Using the SUBTYPE keyword, you name the subtype and provide the name of the base type. The base type may be constrained starting in Oracle8i, but cannot be constrained in earlier releases. You can use the %TYPE attribute on the base type to pick up a data type from a database column or from an existing variable data type. You can also use the %ROWTYPE attribute.

Examples

```
CREATE OR REPLACE PACKAGE mytypes
IS
    SUBTYPE Counter IS INTEGER; -- based on INTEGER type
    TYPE typ_TimeRec IS RECORD (minutes INTEGER, hours
    INTEGER);
    SUBTYPE Time IS typ_TimeRec; -- based on RECORD type
    SUBTYPE ID_Num IS customers.customer_id%TYPE;
    CURSOR cur_cust IS SELECT * FROM customers;
    SUBTYPE CustFile IS cur_cust%ROWTYPE; -- based on cursor
END mytypes;
/
```

Using Subtypes

- Define an identifier that uses the subtype in the declarative section.

```
identifier_name  subtype_name
```

- You can constrain a user-defined subtype when declaring variables of that type.

```
identifier_name  subtype_name(size)
```

- You can constrain a user-defined subtype when declaring the subtype.

ORACLE

Copyright © 2004, Oracle. All rights reserved.

Using Subtypes

After the subtype is declared, you can assign an identifier for that subtype. Subtypes can increase reliability by detecting out-of-range values.

```
DECLARE
  v_rows      mytypes.Counter; --use package subtype dfn
  v_customers mytypes.Counter;
  v_start_time mytypes.Time;
  SUBTYPE      Accumulator IS NUMBER;
  v_total      Accumulator(4,2);
  SUBTYPE      Scale IS NUMBER(1,0);  -- constrained subtype
  v_x_axis      Scale;  -- magnitude range is -9 .. 9
BEGIN
  v_rows := 1;
  v_start_time.minutes := 15;
  v_start_time.hours   := 03;
  dbms_output.put_line('Start time is: ' ||
    v_start_time.hours || ':' || v_start_time.minutes);
END;
/
```

Subtype Compatibility

An unconstrained subtype is interchangeable with its base type.

```
DECLARE
  SUBTYPE Accumulator IS NUMBER;
  v_amount  NUMBER(4,2);
  v_total   Accumulator;
BEGIN
  v_amount := 99.99;
  v_total  := 100.00;
  dbms_output.put_line('Amount is: ' || v_amount);
  dbms_output.put_line('Total is: ' || v_total);
  v_total := v_amount;
  dbms_output.put_line('This works too: ' ||
    v_total);
  -- v_amount := v_amount + 1; Will show value error
END;
/
```

ORACLE

Copyright © 2004, Oracle. All rights reserved.

Type Compatibility

An unconstrained subtype is interchangeable with its base type. Different subtypes are interchangeable if they have the same base type. Different subtypes are also interchangeable if their base types are in the same data type family.

```
DECLARE
  v_rows      mytypes.Counter;
  v_customers mytypes.Counter;
  SUBTYPE Accumulator IS NUMBER;
  v_total     Accumulator(6,2);
BEGIN
  SELECT COUNT(*) INTO v_customers FROM customers;
  SELECT COUNT(*) INTO v_rows FROM orders;
  v_total := v_customers + v_rows;
  DBMS_OUTPUT.PUT_LINE('Total rows from 2 tables: ' ||
    v_total);
EXCEPTION
  WHEN value_error THEN
    DBMS_OUTPUT.PUT_LINE('Error in data type.');
```

```
END;
/
```

Summary

In this lesson, you should have learned how to:

- **Use guidelines for cursor design**
- **Declare, define, and use cursor variables**
- **Use subtypes as data types**

ORACLE

Copyright © 2004, Oracle. All rights reserved.

Summary

- Use the guidelines for designing the cursors
- Take advantage of the features of cursor variables and pass pointers to result sets to different applications.
- You can use subtypes to organize and strongly type data types for an application.

Practice Overview

This practice covers the following topics:

- **Determining the output of a PL/SQL block**
- **Improving the performance of a PL/SQL block**
- **Implementing subtypes**
- **Using cursor variables**

ORACLE

Copyright © 2004, Oracle. All rights reserved.

Practice Overview

In this practice you will determine the output of a PL/SQL code snippet and modify the snippet to improve the performance. Next, you will implement subtypes and use cursor variables to pass values to and from a package.

Practice 2

Note: You will be using oe/oe as the username/password for the practice exercises. Files mentioned in the practice exercises are found in /labs folder. Additionally, solution scripts are provided for each question and are located in the /soln folder. Your instructor will provide you with the exactly location of these files.

1. Determine the output of the following code snippet.

```
SET SERVEROUTPUT ON
BEGIN
  UPDATE orders SET order_status = order_status;
  FOR v_rec IN ( SELECT order_id FROM orders )
  LOOP
    IF SQL%ISOPEN THEN
      DBMS_OUTPUT.PUT_LINE('TRUE - ' || SQL%ROWCOUNT);
    ELSE
      DBMS_OUTPUT.PUT_LINE('FALSE - ' || SQL%ROWCOUNT);
    END IF;
  END LOOP;
END;
/
```

2. Modify the following snippet of code to make better use of the FOR UPDATE clause and improve the performance of the program.

```
DECLARE
  CURSOR cur_update
  IS SELECT * FROM customers
  WHERE credit_limit < 5000 FOR UPDATE;
BEGIN
  FOR v_rec IN cur_update
  LOOP
    IF v_rec IS NOT NULL
    THEN
      UPDATE customers
      SET credit_limit = credit_limit + 200
      WHERE customer_id = v_rec.customer_id;
    END IF;
  END LOOP;
END;
/
```


Practice 2 (continued)

3. Create a package specification that defines subtypes, which can be used for the `warranty_period` field of the `product_information` table. Name this package `MY_TYPES`. The type needs to hold the month and year for a warranty period.
4. Create a package named `SHOW_DETAILS` that contains two subroutines. The first subroutine should show order details for the given `order_id`. The second subroutine should show customer details for the given `customer_id`, including the customer Id, first name, phone numbers, credit limit, and email address. Both the subroutines should use the cursor variable to return the necessary details.

