Exercises for Chapter 16: Records

The Labs below provide you with exercises and suggested answers with discussion related to how those answers resulted. The most important thing to realize is whether your answer works. You should figure out the implications of the answers here and what the effects are from any different answers you may come up with.

Lab 16.1 Record Types

Answer the following questions:

Table-Based and Cursor-Based Records

In this exercise, you will experiment with table-based and cursor-based records. Create the following PL/SQL script:

For Example *ch16_11a.sql*

```
DECLARE
   zip_rec zipcode%ROWTYPE;

BEGIN
   SELECT *
     INTO zip_rec
     FROM zipcode
    WHERE rownum < 2;
END;
```

Answer the following questions:

a) Explain the script created above.

   Answer: The declaration portion of the script contains a declaration of the table-based record, `zip_rec` that has the same structure as a row from the `ZIPCODE` table. The executable portion of the script populates the `zip_rec` record via the `SELECT INTO` statement with a row from the `ZIPCODE` table. Notice that a restriction applied to the `ROWNUM` ensures that the `SELECT INTO` statement always returns a random single row. As mentioned in Chapter 16, there is no need to reference individual record fields when the `SELECT INTO` statement populates the `zip_rec` record because `zip_rec` has a structure identical to a row of the `ZIPCODE` table.

b) Modify the script so that `zip_rec` data is displayed on the screen.

Answer: The script should look similar to the following. Newly added statements are shown in bold.

For Example *ch16_11b.sql*

```
DECLARE
   zip_rec zipcode%ROWTYPE;

BEGIN
   SELECT *
     INTO zip_rec
     FROM zipcode
    WHERE rownum < 2;

   DBMS_OUTPUT.PUT_LINE ('Zip:           '||zip_rec.zip);
   DBMS_OUTPUT.PUT_LINE ('City:          '||zip_rec.city);
   DBMS_OUTPUT.PUT_LINE ('State:         '||zip_rec.state);
   DBMS_OUTPUT.PUT_LINE ('Created By:    '||zip_rec.created_by);
   DBMS_OUTPUT.PUT_LINE ('Created Date:  '||zip_rec.created_date);
   DBMS_OUTPUT.PUT_LINE ('Modified By:   '||zip_rec.modified_by);
   DBMS_OUTPUT.PUT_LINE ('Modified Date: '||zip_rec.modified_date);
END;
```

When run, this version of the script produces output as follows:

```
Zip:           12345
City:          New York
State:         NY
Created By:    STUDENT
Created Date:  11/06/2014 13:08
Modified By:   STUDENT
Modified Date: 11/06/2014 13:08
```

c) Modify the script created in the previous exercise (ch16_11b.sql) so that `zip_rec` is defined as a cursor-based record.

Answer: The script should look similar to the following script. Changes are shown in bold.

For Example *ch16_11c.sql*

```
DECLARE
   CURSOR zip_cur IS
      SELECT *
        FROM zipcode
       WHERE rownum < 4;

   zip_rec zip_cur%ROWTYPE;
BEGIN
   OPEN zip_cur;
   LOOP
      FETCH zip_cur INTO zip_rec;
      EXIT WHEN zip_cur%NOTFOUND;

      DBMS_OUTPUT.PUT_LINE ('Zip:           '||zip_rec.zip);
      DBMS_OUTPUT.PUT_LINE ('City:          '||zip_rec.city);
      DBMS_OUTPUT.PUT_LINE ('State:         '||zip_rec.state);
      DBMS_OUTPUT.PUT_LINE ('Created By:    '||zip_rec.created_by);
      DBMS_OUTPUT.PUT_LINE ('Created Date:  '||zip_rec.created_date);
      DBMS_OUTPUT.PUT_LINE ('Modified By:   '||zip_rec.modified_by);
      DBMS_OUTPUT.PUT_LINE ('Modified Date: '||zip_rec.modified_date);
   END LOOP;
END;
```

The declaration portion of the script contains a definition of the `zip_cur` cursor that returns three records from the `ZIPCODE` table. In this case, the number of records returned by the cursor has been chosen for one reason only, so that the cursor loop iterates more than once. Next, it contains the definition of the cursor-based record, `zip_rec`.

The executable portion of the script populates the `zip_rec` record and displays its data on the screen via the simple cursor loop.

This version of the script produces the following output:

```
Zip:           12345
City:          New York
State:         NY
Created By:    STUDENT
Created Date:  11/06/2014 13:08
Modified By:   STUDENT
Modified Date: 11/06/2014 13:08
Zip:           00914
City:          Santurce
State:         PR
Created By:    AMORRISO
Created Date:  08/03/2007 00:00
Modified By:   ARISCHER
Modified Date: 11/24/2007 00:00
Zip:           01247
City:          North Adams
State:         MA
Created By:    AMORRISO
Created Date:  08/03/2007 00:00
Modified By:   ARISCHER
Modified Date: 11/24/2007 00:00
```

d) Modify the script created in the previous exercise (ch16_11c.sql). Change the structure of the `zip_rec` record so that it contains total number of students in a given city, state, and ZIP code. Do not include audit columns such as `CREATED_BY` and `CREATED_DATE` in the record structure.

Answer: This version of the script should look similar to the following script. All changes are shown in bold.

For Example *ch16_11d.sql*

```
DECLARE
   CURSOR zip_cur IS
      SELECT city, state, z.zip, COUNT(*) students
        FROM zipcode z, student s
```

```
       WHERE z.zip = s.zip
      GROUP BY city, state, z.zip;

   zip_rec zip_cur%ROWTYPE;
BEGIN
   OPEN zip_cur;
   LOOP
      FETCH zip_cur INTO zip_rec;
      EXIT WHEN zip_cur%NOTFOUND;

      DBMS_OUTPUT.PUT_LINE ('Zip:      '||zip_rec.zip);
      DBMS_OUTPUT.PUT_LINE ('City:     '||zip_rec.city);
      DBMS_OUTPUT.PUT_LINE ('State:    '||zip_rec.state);
      DBMS_OUTPUT.PUT_LINE ('Students: '||zip_rec.students);
  END LOOP;
END;
```

In this example, the cursor `SELECT` statement has been modified so that it returns total number of students for a given city, state, and zip code. Notice that the `ROWNUM` restriction has been removed so that the total number of students is calculated correctly.

Note that if you run this script in SQL*Plus, you may need to increase the buffer size so that the script does not cause a buffer overflow error.

Consider the partial output retuned by this example:

```
Zip:      06483
City:     Oxford
State:    CT
Students: 1
Zip:      06902
City:     Stamford
State:    CT
Students: 1
Zip:      07055
City:     Passaic
State:    NJ
Students: 2
…
```

Next, assume that just like in the previous version of the script (ch16_11c.sql), you would like to display only four records on the screen. This can be achieved as follows:

For Example *ch16_11e.sql*

```
DECLARE
   CURSOR zip_cur IS
      SELECT city, state, z.zip, COUNT(*) students
        FROM zipcode z, student s
       WHERE z.zip = s.zip
      GROUP BY city, state, z.zip;

   zip_rec zip_cur%ROWTYPE;
   v_counter INTEGER := 0;
BEGIN
   OPEN zip_cur;
   LOOP
```

```
         FETCH zip_cur INTO zip_rec;
         EXIT WHEN zip_cur%NOTFOUND;

         v_counter := v_counter + 1;

         IF v_counter <= 4
         THEN
            DBMS_OUTPUT.PUT_LINE ('Zip:      '||zip_rec.zip);
            DBMS_OUTPUT.PUT_LINE ('City:     '||zip_rec.city);
            DBMS_OUTPUT.PUT_LINE ('State:    '||zip_rec.state);
            DBMS_OUTPUT.PUT_LINE ('Students: '||zip_rec.students);
         END IF;
      END LOOP;
   END;
```

User-Defined Records

In this exercise, you will investigate user-defined records. Create the following PL/SQL script:

For Example *ch16_12a.sql*

```
DECLARE
   CURSOR zip_cur IS
      SELECT zip, COUNT(*) students
        FROM student
      GROUP BY zip;

   TYPE zip_info_type IS RECORD
      (zip_code VARCHAR2(5)
      ,students INTEGER);

   zip_info_rec zip_info_type;
BEGIN
   FOR zip_rec IN zip_cur
   LOOP
      zip_info_rec.zip_code := zip_rec.zip;
      zip_info_rec.students := zip_rec.students;
   END LOOP;
END;
```

Answer the following questions:

a) Explain the script ch16_12a.sql.

Answer: The declaration portion of the script contains `zip_cur` cursor, which returns total number of students corresponding to a particular ZIP code. Next, it contains the declaration of the user-defined record type, `zip_info_type`, which has two fields, and the actual user-defined record, `zip_info_rec`. The executable portion of the script populates the `zip_info_rec` record via the cursor `FOR LOOP`. As mentioned earlier, because `zip_info_rec` is a user-defined record, each record field is assigned a value individually.

b) Modify the script so that `zip_info_rec` data is displayed on the screen only for the first five records returned by the `zip_cur` cursor.

Answer: The script should look similar to the following script. Newly added statements are shown in bold.

For Example *ch16_12b.sql*

```
DECLARE
   CURSOR zip_cur IS
      SELECT zip, COUNT(*) students
        FROM student
      GROUP BY zip;

   TYPE zip_info_type IS RECORD
      (zip_code VARCHAR2(5)
      ,students INTEGER);

   zip_info_rec zip_info_type;
   v_counter INTEGER := 0;
BEGIN
   FOR zip_rec IN zip_cur
   LOOP
      zip_info_rec.zip_code := zip_rec.zip;
      zip_info_rec.students := zip_rec.students;

      v_counter := v_counter + 1;
      IF v_counter <= 5
      THEN
         DBMS_OUTPUT.PUT_LINE ('Zip Code: '||zip_info_rec.zip_code);
         DBMS_OUTPUT.PUT_LINE ('Students: '||zip_info_rec.students);
         DBMS_OUTPUT.PUT_LINE ('--------------------');
      END IF;
   END LOOP;
END;
```

In order to display information for the first five records returned by the `zip_cur` cursor, a new variable, `v_counter`, is declared. For each iteration of the loop, the value of this variable is incremented by one. As long as the value of the variable `v_counter` is less than or equal to five, the data of the `zip_info_rec` record is displayed on the screen.

When run, this script produces the following output:

```
Zip Code: 01247
Students: 1
--------------------
Zip Code: 02124
Students: 1
--------------------
Zip Code: 02155
Students: 1
--------------------
Zip Code: 02189
Students: 1
--------------------
Zip Code: 02563
Students: 1
--------------------
```

c) Modify the script created in the previous exercise (ch16_12b.sql). Change the structure of the `zip_info_rec` record so that it also contains total number of instructors for a given zip code. Populate this new record and display its data on the screen for the first five records returned by the `zip_cur` cursor.

Answer: The script should look similar to the following script. Changes are shown in bold.

For Example *ch16_12c.sql*

```
DECLARE
   CURSOR zip_cur IS
      SELECT zip
        FROM zipcode
       WHERE ROWNUM <= 5;

   TYPE zip_info_type IS RECORD
      (zip_code    VARCHAR2(5)
      ,students    INTEGER
      ,instructors INTEGER);

   zip_info_rec zip_info_type;
BEGIN
   FOR zip_rec IN zip_cur
   LOOP
      zip_info_rec.zip_code := zip_rec.zip;

      SELECT COUNT(*)
        INTO zip_info_rec.students
        FROM student
       WHERE zip = zip_info_rec.zip_code;

      SELECT COUNT(*)
        INTO zip_info_rec.instructors
        FROM instructor
       WHERE zip = zip_info_rec.zip_code;

      DBMS_OUTPUT.PUT_LINE ('Zip Code:    '||zip_info_rec.zip_code);
      DBMS_OUTPUT.PUT_LINE ('Students:    '||zip_info_rec.students);
      DBMS_OUTPUT.PUT_LINE ('Instructors: '||zip_info_rec.instructors);
      DBMS_OUTPUT.PUT_LINE ('--------------------');
   END LOOP;
END;
```

Consider the changes applied to this version of the script. In the declaration portion of the script, the cursor `SELECT` statement has changed so that records are retrieved from the `ZIPCODE` table rather than the `STUDENT` table. This change allows you to see accurately the total number of students and instructors in a particular ZIP code. In addition, because the cursor `SELECT` statement does not have group function, the `ROWNUM` restriction is listed in the `WHERE` clause so that only the first five records are returned. The structure of the user-defined record type, `zip_info_type`, has changed so that total number of instructors for a given ZIP code is stored in the `instructors` field.

In the executable portion of the script, there are two `SELECT INTO` statements that populate `zip_info_rec.students` and `zip_info_rec.instructors` fields, respectively.

When run, this example produces the following output:

```
Zip Code:    00914
Students:    0
Instructors: 0
--------------------
Zip Code:    01247
Students:    1
Instructors: 0
--------------------
Zip Code:    02124
Students:    1
Instructors: 0
--------------------
Zip Code:    02155
Students:    1
Instructors: 0
--------------------
Zip Code:    02189
Students:    1
Instructors: 0
--------------------
```

Consider another version of the same script. Here, instead of using two `SELECT INTO` statements to calculate the total number of students and instructors in a particular ZIP code, the cursor `SELECT` statement contains outer joins.

For Example *ch16_12d.sql*

```
DECLARE
   CURSOR zip_cur IS
      SELECT z.zip, COUNT(student_id) students, COUNT(instructor_id) instructors
        FROM zipcode z, student s, instructor i
       WHERE z.zip = s.zip (+)
         AND z.zip = i.zip (+)
      GROUP BY z.zip;

   TYPE zip_info_type IS RECORD
      (zip_code    VARCHAR2(5)
      ,students    INTEGER
      ,instructors INTEGER);

   zip_info_rec zip_info_type;
   v_counter INTEGER := 0;
BEGIN
   FOR zip_rec IN zip_cur
   LOOP
      zip_info_rec.zip_code    := zip_rec.zip;
      zip_info_rec.students    := zip_rec.students;
      zip_info_rec.instructors := zip_rec.instructors;

      v_counter := v_counter + 1;
      IF v_counter <= 5
      THEN
         DBMS_OUTPUT.PUT_LINE ('Zip Code:    '||zip_info_rec.zip_code);
         DBMS_OUTPUT.PUT_LINE ('Students:    '||zip_info_rec.students);
         DBMS_OUTPUT.PUT_LINE ('Instructors: '||zip_info_rec.instructors);
         DBMS_OUTPUT.PUT_LINE ('--------------------');
      END IF;
   END LOOP;
END;
```

Lab 16.2 Nested Records

In this exercise, you will experiment with nested records. Create the following PL/SQL script:

For Example *ch16_13a.sql*

```
DECLARE
   TYPE last_name_type IS TABLE OF student.last_name%TYPE INDEX BY PLS_INTEGER;

   TYPE zip_info_type IS RECORD
      (zip           VARCHAR2(5)
      ,last_name_tab last_name_type);

   CURSOR name_cur (p_zip VARCHAR2) IS
      SELECT last_name
        FROM student
       WHERE zip = p_zip;

   zip_info_rec zip_info_type;
   v_zip        VARCHAR2(5) := '&sv_zip';
   v_counter    INTEGER := 0;
BEGIN
   zip_info_rec.zip := v_zip;

   FOR name_rec IN name_cur (v_zip)
   LOOP
      v_counter := v_counter + 1;
      zip_info_rec.last_name_tab(v_counter) := name_rec.last_name;
   END LOOP;
END;
```

Answer the following questions:

a) Explain the script ch16_13a.sql.

Answer: The declaration portion of the script contains associative array (index-by table) type, `last_name_type`, record type, `zip_info_type`, and nested-user-defined record, `zip_info_rec`, declarations. The field, `last_name_tab`, of the `zip_info_rec` is an associative array that is populated with the help of the cursor, `name_cur`. In addition, the declaration portion also contains two variables, `v_zip` and `v_counter`. The variable `v_zip` is used to store incoming value of the ZIP code provided at runtime. The variable `v_counter` is used to populate the associative array, `last_name_tab`.

The executable portion of the script assigns values to the individual record fields, `zip` and `last_name_tab`. As mentioned previously, the `last_name_tab` is an associative array, and it is populated via cursor `FOR LOOP`.

b) Modify the script so that `zip_info_rec` data is displayed on the screen. Make sure that a value of the ZIP code is displayed only once. Provide the value of ‘11368’ when running the script.

Answer: The new version of the script should look similar to the following. Newly added statements are highlighted in bold.

For Example *ch16_13b.sql*

```
DECLARE
   TYPE last_name_type IS TABLE OF student.last_name%TYPE INDEX BY PLS_INTEGER;

   TYPE zip_info_type IS RECORD
      (zip          VARCHAR2(5)
      ,last_name_tab last_name_type);

   CURSOR name_cur (p_zip VARCHAR2) IS
      SELECT last_name
        FROM student
       WHERE zip = p_zip;

   zip_info_rec zip_info_type;
   v_zip        VARCHAR2(5) := '&sv_zip';
   v_counter    INTEGER := 0;
BEGIN
   zip_info_rec.zip := v_zip;
   DBMS_OUTPUT.PUT_LINE ('Zip: '||zip_info_rec.zip);

   FOR name_rec IN name_cur (v_zip)
   LOOP
      v_counter := v_counter + 1;
      zip_info_rec.last_name_tab(v_counter) := name_rec.last_name;

      DBMS_OUTPUT.PUT_LINE ('Names('||v_counter||'): '||
         zip_info_rec.last_name_tab(v_counter));
   END LOOP;
END;
```

In order to display the value of the zip code only once, the `DBMS_OUTPUT.PUT_LINE` statement

```
DBMS_OUTPUT.PUT_LINE ('Zip: '||zip_info_rec.zip);
```

is placed outside the loop.

When run, this script produces the following output:

```
Zip: 11368
Names(1): Lasseter
Names(2): Miller
Names(3): Boyd
Names(4): Griffen
Names(5): Hutheesing
Names(6): Chatman
```

c) Modify the script created in the previous exercise (ch16_13b.sql). Instead of providing a value for a ZIP code at runtime, populate it via the cursor `FOR LOOP`. The `SELECT` statement associated with the new cursor should return ZIP codes that have more than one student in them.

Answer: The script should look similar to the following script. Changes are shown in bold.

For Example *ch16_13c.sql*

```
DECLARE
   TYPE last_name_type IS TABLE OF student.last_name%TYPE INDEX BY PLS_INTEGER;

   TYPE zip_info_type IS RECORD
      (zip          VARCHAR2(5)
      ,last_name_tab last_name_type);

   CURSOR zip_cur IS
      SELECT zip, COUNT(*)
        FROM student
      GROUP BY zip
      HAVING COUNT(*) > 1;

   CURSOR name_cur (p_zip VARCHAR2) IS
      SELECT last_name
        FROM student
       WHERE zip = p_zip;

   zip_info_rec zip_info_type;
   v_counter    INTEGER;
BEGIN
   FOR zip_rec IN zip_cur
   LOOP
      zip_info_rec.zip := zip_rec.zip;
      DBMS_OUTPUT.PUT_LINE ('Zip: '||zip_info_rec.zip);

      v_counter := 0;
      FOR name_rec IN name_cur (zip_info_rec.zip)
      LOOP
         v_counter := v_counter + 1;
         zip_info_rec.last_name_tab(v_counter) := name_rec.last_name;

         DBMS_OUTPUT.PUT_LINE ('Names('||v_counter||'): '||
            zip_info_rec.last_name_tab(v_counter));
      END LOOP;
      DBMS_OUTPUT.PUT_LINE ('----------');
   END LOOP;
END;
```

In the preceding script, you declared a new cursor called `zip_cur`. This cursor returns ZIP codes that have more than one student in them. Next, in the body of the script, you use nested cursors to populate the `last_name_tab` associative array for each value of ZIP code. First, the outer cursor `FOR LOOP` populates the `zip` field of the `zip_info_rec` and displays its value on the screen. Then it passes the `zip` field as a parameter to the inner cursor `FOR LOOP` that populates `last_name_tab` table with last names of corresponding students.

Consider the partial output of the preceding example:

```
Zip: 06820
Names(1): Scrittorale
Names(2): Padel
Names(3): Kiraly
----------
Zip: 06830
Names(1): Dennis
Names(2): Meshaj
Names(3): Dalvi
----------
Zip: 06880
Names(1): Cheevens
Names(2): Miller
----------
…
```

Lab 16.3 Collections of Records

In this exercise, you will investigate collections of records. Answer the following questions:

a) Modify the script ch16_9a.sql used in Chapter 16. Instead of using associative array, use a varray.

   Answer: The newly created script should look similar to the following. All changes are highlighted in bold.

   For Example *ch16_9c.sql*

```
DECLARE
   CURSOR name_cur IS
      SELECT first_name, last_name
        FROM student
       WHERE ROWNUM <= 4;

   TYPE name_type IS VARRAY(4) OF name_cur%ROWTYPE;

   name_tab name_type := name_type();
   v_index INTEGER := 0;
BEGIN
   FOR name_rec IN name_cur
   LOOP
      v_index := v_index + 1;
      name_tab.EXTEND;

      name_tab(v_index).first_name := name_rec.first_name;
      name_tab(v_index).last_name  := name_rec.last_name;

      DBMS_OUTPUT.PUT_LINE('First Name('||v_index ||'): '||
         name_tab(v_index).first_name);
      DBMS_OUTPUT.PUT_LINE('Last Name('||v_index ||'): '||
         name_tab(v_index).last_name);
   END LOOP;
```

```
END;
```

In this version of the script, the `name_tab` collection variable is declared as a varray with four elements. Note that in this version, the collection is initialized and its size is incremented before it is populated with the new record.

This version of the script produces the output identical to the original example:

```
First Name(1): George
Last Name(1): Kocka
First Name(2): Janet
Last Name(2): Jung
First Name(3): Kathleen
Last Name(3): Mulroy
First Name(4): Joel
Last Name(4): Brendler
```

b) Modify the script created in the previous exercise (ch16_9c.sql). Replace cursor-based record with user-defined record.

Answer: The version of the script should look similar to the following script. Modifications are shown in bold.

For Example *ch16_9d.sql*

```
DECLARE
   CURSOR name_cur IS
      SELECT first_name, last_name
        FROM student
       WHERE ROWNUM <= 4;

   TYPE name_rec_type IS RECORD
      (first_name  VARCHAR2(15)
      ,last_name   VARCHAR2(30));

   TYPE name_type IS VARRAY(4) OF name_rec_type;

   name_rec name_rec_type;
   name_tab name_type := name_type();
   v_index  INTEGER := 0;
BEGIN
   FOR rec IN name_cur
   LOOP
      name_rec := rec;
      v_index := v_index + 1;
      name_tab.EXTEND;

      name_tab(v_index).first_name := name_rec.first_name;
      name_tab(v_index).last_name  := name_rec.last_name;

      DBMS_OUTPUT.PUT_LINE('First Name('||v_index ||'): '||
         name_tab(v_index).first_name);
      DBMS_OUTPUT.PUT_LINE('Last Name('||v_index ||'): '||
         name_tab(v_index).last_name);
   END LOOP;
```

```
END;
```

This version of the script contains a new record type, `name_rec_type`, and the corresponding user-defined record variable, `name_rec`. As a result, a cursor record, `rec`, implicitly defined by the cursor `FOR LOOP` is assigned to the user-defined record, `name_rec`, Note that the rest of the script remains unchanged.

When run, this script produces output identical to the previous versions:

```
First Name(1): George
Last Name(1): Kocka
First Name(2): Janet
Last Name(2): Jung
First Name(3): Kathleen
Last Name(3): Mulroy
First Name(4): Joel
Last Name(4): Brendler
```

Next, consider slightly modified version of the script that does not have user-defined record variable, `name_rec`. Affected statements are shown in bold.

For Example *ch16_9d.sql*

```
DECLARE
   CURSOR name_cur IS
      SELECT first_name, last_name
        FROM student
       WHERE ROWNUM <= 4;

   TYPE name_rec_type IS RECORD
      (first_name  VARCHAR2(15)
      ,last_name   VARCHAR2(30));

   TYPE name_type IS VARRAY(4) OF name_rec_type;

   name_tab name_type := name_type();
   v_index  INTEGER := 0;
BEGIN
   FOR rec IN name_cur
   LOOP
      v_index := v_index + 1;
      name_tab.EXTEND;

      name_tab(v_index).first_name := rec.first_name;
      name_tab(v_index).last_name  := rec.last_name;

      DBMS_OUTPUT.PUT_LINE('First Name('||v_index ||'): '||
         name_tab(v_index).first_name);
      DBMS_OUTPUT.PUT_LINE('Last Name('||v_index ||'): '||
         name_tab(v_index).last_name);
   END LOOP;
END;
```

Try It Yourself

The projects in this section are meant to have you use all of the skills that you have acquired throughout this chapter. Here are some exercises that will help you test the depth of your understanding.

1) Create an associative array with the element type of a user-defined record. This record should contain first name, last name, and the total number of courses that a particular instructor teaches. Display the records of the associative array on the screen.

   Answer: The script should look similar to the following:

   For Example *ch16_14a.sql*

```
DECLARE
   CURSOR instructor_cur IS
      SELECT first_name, last_name, COUNT(UNIQUE s.course_no) courses
        FROM instructor i
        LEFT OUTER JOIN section s
          ON (s.instructor_id = i.instructor_id)
      GROUP BY first_name, last_name;

   TYPE rec_type IS RECORD
      (first_name     INSTRUCTOR.FIRST_NAME%TYPE
      ,last_name      INSTRUCTOR.LAST_NAME%TYPE
      ,courses_taught NUMBER);

   TYPE instructor_type IS TABLE OF REC_TYPE INDEX BY PLS_INTEGER;

   instructor_tab instructor_type;

   v_index INTEGER := 0;
BEGIN
   FOR instructor_rec IN instructor_cur
   LOOP
      v_index := v_index + 1;

      -- Populate associative array of records
      instructor_tab(v_index).first_name     := instructor_rec.first_name;
      instructor_tab(v_index).last_name      := instructor_rec.last_name;
      instructor_tab(v_index).courses_taught := instructor_rec.courses;

      DBMS_OUTPUT.PUT_LINE ('Instructor, '||
         instructor_tab(v_index).first_name||' '||
         instructor_tab(v_index).last_name||', teaches '||
         instructor_tab(v_index).courses_taught||' courses.');
   END LOOP;
END;
```

   In this script, you define a cursor against the `INSTRUCTOR` and `SECTION` tables that is used to populate the associative array of records, `instructor_tab`. Each row of this table is a user-defined record of three elements. You populate the associative array via the cursor `FOR LOOP`. Consider the notation used to reference each record element of the associative array:

```
instructor_tab(v_counter).first_name
instructor_tab(v_counter).last_name
instructor_tab(v_counter).courses_taught
```

To reference each row of the associative array, you use the counter variable. However, because each row of this table is a record, you must also reference individual fields of the underlying record. When run, this script produces the following output:

```
Instructor, Fernand Hanks, teaches 9 courses.
Instructor, Charles Lowry, teaches 9 courses.
Instructor, Rick Chow, teaches 0 courses.
Instructor, Nina Schorin, teaches 10 courses.
Instructor, Gary Pertez, teaches 10 courses.
Instructor, Anita Morris, teaches 10 courses.
Instructor, Marilyn Frantzen, teaches 9 courses.
Instructor, Irene Willig, teaches 0 courses.
Instructor, Tom Wojick, teaches 10 courses.
Instructor, Todd Smythe, teaches 10 courses.
```

2) Modify the script created in previous exercise (exercise 1 above). Instead of using an associative array, use a nested table.

Answer: The script should look similar to the following. All changes are highlighted in bold.

For Example *ch16_14b.sql*

```
DECLARE
   CURSOR instructor_cur IS
      SELECT first_name, last_name, COUNT(UNIQUE s.course_no) courses
        FROM instructor i
        LEFT OUTER JOIN section s
          ON (s.instructor_id = i.instructor_id)
      GROUP BY first_name, last_name;

   TYPE rec_type IS RECORD
      (first_name     INSTRUCTOR.FIRST_NAME%TYPE
      ,last_name      INSTRUCTOR.LAST_NAME%TYPE
      ,courses_taught NUMBER);

   TYPE instructor_type IS TABLE OF REC_TYPE;
   instructor_tab instructor_type := instructor_type();

   v_index INTEGER := 0;
BEGIN
   FOR instructor_rec IN instructor_cur
   LOOP
      v_index := v_index + 1;
      instructor_tab.EXTEND;

      -- Populate nested table of records
      instructor_tab(v_index).first_name     := instructor_rec.first_name;
      instructor_tab(v_index).last_name      := instructor_rec.last_name;
      instructor_tab(v_index).courses_taught := instructor_rec.courses;

      DBMS_OUTPUT.PUT_LINE ('Instructor, '||
         instructor_tab(v_index).first_name||' '||
         instructor_tab(v_index).last_name||', teaches '||
         instructor_tab(v_index).courses_taught||' courses.');
   END LOOP;
END;
```

Notice that the `instructor_tab` must be initialized and extended before its individual elements can be referenced.

3) Modify the script created in previous exercise (exercise 2 above). Instead of using a nested table, use a varray.

   Answer: The version of the script should look similar to the following. Affected statements are highlighted in bold.

   For Example *ch16_14c.sql*

```
DECLARE
   CURSOR instructor_cur IS
      SELECT first_name, last_name, COUNT(UNIQUE s.course_no) courses
        FROM instructor i
        LEFT OUTER JOIN section s
          ON (s.instructor_id = i.instructor_id)
      GROUP BY first_name, last_name;

   TYPE rec_type IS RECORD
      (first_name     INSTRUCTOR.FIRST_NAME%TYPE
      ,last_name      INSTRUCTOR.LAST_NAME%TYPE
      ,courses_taught NUMBER);

   TYPE instructor_type IS VARRAY(10) OF REC_TYPE;
   instructor_tab instructor_type := instructor_type();

   v_index INTEGER := 0;
BEGIN
   FOR instructor_rec IN instructor_cur
   LOOP
      v_index := v_index + 1;
      instructor_tab.EXTEND;

      -- Populate varray of records
      instructor_tab(v_index).first_name     := instructor_rec.first_name;
      instructor_tab(v_index).last_name      := instructor_rec.last_name;
      instructor_tab(v_index).courses_taught := instructor_rec.courses;

      DBMS_OUTPUT.PUT_LINE ('Instructor, '||
         instructor_tab(v_index).first_name||' '||
         instructor_tab(v_index).last_name||', teaches '||
         instructor_tab(v_index).courses_taught||' courses.');
   END LOOP;
END;
```

This version of the script is almost identical to the previous version. Instead of using a nested table, you are using a varray of 10 elements.

4) Create a user-defined record with four fields: `course_no`, `description`, `cost`, and `prerequisite_rec`. The last field, `prerequisite_rec`, should be a user-defined record with three fields: `prereq_no`, `prereq_desc`, and `prereq_cost`. For any ten courses that have a prerequisite course, populate the user-defined record with all corresponding data and display its information on the screen.

   Answer: The script should look similar to the following:

   For Example *ch16_15a.sql*

```
DECLARE
   CURSOR c_cur IS
      SELECT course_no, description, cost, prerequisite
        FROM course
       WHERE prerequisite IS NOT NULL
         AND rownum <= 10;

   TYPE prerequisite_type IS RECORD
      (prereq_no   NUMBER
      ,prereq_desc VARCHAR(50)
      ,prereq_cost NUMBER);

   TYPE course_type IS RECORD
      (course_no        NUMBER
      ,description      VARCHAR2(50)
      ,cost             NUMBER
      ,prerequisite_rec PREREQUISITE_TYPE);

   course_rec COURSE_TYPE;
BEGIN
   FOR c_rec in c_cur
   LOOP
      course_rec.course_no   := c_rec.course_no;
      course_rec.description := c_rec.description;
      course_rec.cost        := c_rec.cost;

      SELECT course_no, description, cost
        INTO course_rec.prerequisite_rec.prereq_no,
             course_rec.prerequisite_rec.prereq_desc,
             course_rec.prerequisite_rec.prereq_cost
        FROM course
       WHERE course_no = c_rec.prerequisite;

      DBMS_OUTPUT.PUT_LINE ('Course: '||
         course_rec.course_no||' - '||course_rec.description);
      DBMS_OUTPUT.PUT_LINE ('Cost: '|| course_rec.cost);
      DBMS_OUTPUT.PUT_LINE ('Prerequisite: '||
         course_rec.prerequisite_rec. prereq_no||' - '||
         course_rec.prerequisite_rec.prereq_desc);
      DBMS_OUTPUT.PUT_LINE ('Prerequisite Cost: '||
         course_rec.prerequisite_rec.prereq_cost);
      DBMS_OUTPUT.PUT_LINE ('========================================');
   END LOOP;
END;
```

In the declaration portion of the script, you define a cursor against the `COURSE` table; two user-defined record types, `prerequisite_type` and `course_type`; and user-defined record, `course_rec`. It is important to note the order in which the record types are declared. The `prerequsite_type` must be declared first because one of the `course_type` elements is of the `prerequisite_type`.

In the executable portion of the script, you populate `course_rec` via the cursor `FOR LOOP`. First, you assign values to the `course_rec.course_no`, `course_rec.description`, and `course_rec.cost`. Next, you populate the nested record, `prerequsite_rec`, via the `SELECT INTO` statement against the `COURSE` table.

Consider the notation used to reference individual elements of the nested record:

```
course_rec.prerequisite_rec.prereq_no
course_rec.prerequisite_rec.prereq_desc
course_rec.prerequisite_rec.prereq_cost
```

You specify the name of the outer record followed by the name of the inner (nested) record followed by the name of the element. Finally, you display record information on the screen. Note that this script does not contain a `NO_DATA_FOUND` exception handler even though there is a `SELECT INTO` statement. Why do you think this is the case?

When run, the script produces the following output:

```
Course: 230 - Intro to the Internet
Cost: 1095
Prerequisite: 10 - Technology Concepts
Prerequisite Cost: 1195
========================================
Course: 100 - Hands-On Windows
Cost: 1195
Prerequisite: 20 - Intro to Information Systems
Prerequisite Cost: 1195
========================================
Course: 140 - Systems Analysis
Cost: 1195
Prerequisite: 20 - Intro to Information Systems
Prerequisite Cost: 1195
========================================
Course: 142 - Project Management
Cost: 1195
Prerequisite: 20 - Intro to Information Systems
Prerequisite Cost: 1195
========================================
Course: 147 - GUI Design Lab
Cost: 1195
Prerequisite: 20 - Intro to Information Systems
Prerequisite Cost: 1195
========================================
Course: 204 - Intro to SQL
Cost: 1195
Prerequisite: 20 - Intro to Information Systems
Prerequisite Cost: 1195
```

```
========================================
Course: 240 – Intro to the BASIC Language
Cost: 1095
Prerequisite: 25 – Intro to Programming
Prerequisite Cost: 1195
========================================
Course: 420 – Database System Principles
Cost: 1195
Prerequisite: 25 – Intro to Programming
Prerequisite Cost: 1195
========================================
Course: 120 – Intro to Java Programming
Cost: 1195
Prerequisite: 80 – Programming Techniques
Prerequisite Cost: 1595
========================================
Course: 220 – PL/SQL Programming
Cost: 1195
Prerequisite: 80 – Programming Techniques
Prerequisite Cost: 1595
========================================
```