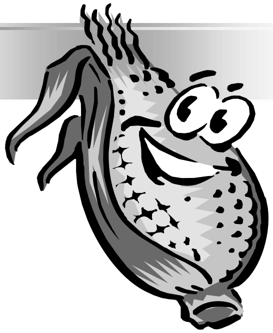
8장 추가

Oracle PL/SQL Programming



PL/SQL

- Concepts
  - a procedural programming language extended from SQL
  - 4GL(4th Generation Language), Ada-like syntax
- Blocks
  - a PL/SQL program is structured using distinct blocks
  - three sections of a block
    - *declaration section* : declare all variables, constants, exceptions, etc.
    - *main program body* : executable statements for the block
    - *exception handling section* : exception handler for the block
  - blocks can be nested

A Simple PL/SQL Program

```
PROCEDURE update_part_unitprice (part_id IN INTEGER, new_price IN
  NUMBER)
   IS
     invalid_part EXCEPTION;
   BEGIN
   -- HERE’S AN UPDATE STATEMENT TO UPDATE A DB RECORD
   UPDATE sales.parts
     SET unit_price = new_price
     WHERE id = part_id;
   -- HERE’S AN ERROR-CHECKING STATEMENT
   IF SQL%NOTFOUND THEN
     RAISE invalid_part;
   END IF;
   -- HERE’S AN ERROR-HANDLING ROUTINE
   WHEN invalid_part THEN
     raise_application_error(-20000, ‘Invalid Part ID’);
   END update_part_unitprice;
```

Data Types

- Commenting
  - -- : a single-line comment
  - /* … */ : a multi-line comment
- Variable and constant declaration

```
DECLARE
  emp_id INTEGER;
  standard_commission CONSTANT INTEGER := 500;
  counter INTEGER := 0; -- initialization
  emp_commission INTEGER DEFAULT 0; -- default value
  … …
```

- Data types and subtypes
  - a subtype is a constrained version of its base type
  - supporting Oracle and ANSI/ISO datatypes

Datatype and Subtype (1/2)

- BINARY_INTEGER
  - subtypes : NATURAL, NATURALN(no NULLs), POSITIVE, POSITIVEN(no NULLs), SIGNTYPE(only –1, 0, 1)
  - signed integers
- NUMBER(precision, scale)
  - subtypes : DEC, DECIMAL, DOUBLE PRECISION, INTEGER, INT, FLOAT(precision), NUMERIC, REAL, SMALLINT
- CHAR(size)
  - subtype : CHARACTER(size) [size = 1 ~ 32767]
  - fixed-length character strings. maximum bytes is 2000
- VARCHAR2(size)
  - subtype : VARCHAR(size) [size = 1 ~ 32767], STRING
  - variable-length character string. maximum bytes is 4000

Datatype and Subtype (2/2)

- DATE
  - time-related information including dates, hours, minutes, sec.
- BOOLEAN : TRUE, FALSE, NULL
- CLOB/BLOB/BFILE
- User-defined subtypes
  - customizing the acceptable domain of values for variables
  - cannot define constrained subtypes directly
  - a subtype is interchangeable with its base type

```
DECLARE
  varchar2_50 VARCHAR2(50); -- constrained datatype
  SUBTYPE description IS Varchar2_50;
  current_description description DEFAULT ‘Unknown;
  …
BEGIN
  current_description := varchar2_50;
  …
```

User-Defined Composite Types (1/3)

- Records
  - a group of related fields, like a tuple in a table

```
TYPE part_record IS RECORD (
  id INTEGER,
  unit_price NUMBER(10,2),
  description VARCHAR2(200)
);
current_part part_record;
```

- Nested tables
  - an *unlimited* number of rows, like tables in a database

```
TYPE parts_table IS TABLE OF part_record;
current_parts_table parts_table;
…
```

User-Defined Composite Types (2/3)

- Varying arrays
  - a *limited* number of rows, like a table in a database

```
TYPE parts_varying_arr IS VARRAY(3) OF part_record;
current_parts_table2 Parts_Varying_Arr;
…
```

- Attributes
  - %TYPE : capturing the datatype of another program construct or column in a database table at runtime
  - %ROWTYPE : can reference types of record variables and other constructs at runtime
  - *simplifying* the declaration of program constructs, and making programs *flexible* to database modifications

User-Defined Composite Types (3/3)

```
DECLARE
  TYPE part_record IS RECORD (
    id sales.parts.id%TYPE,
    unit_price sales.parts.unit_price%TYPE,
    description sales.parts.description%TYPE
  );
  current_part part_record;

  TYPE parts_table IS TABLE OF sales.parts%ROWTYPE;
  current_table parts_table;
```

Cursors

- Cursor
  - a work area for a SQL statement
  - cursor declaration

```
CURSOR parts_cursor IS
  SELECT * FROM sales.parts;

CURSOR customers_cursor (state_id CHAR) IS
  SELECT id, last_name, first_name, phone
  FROM sales.customers
  WHERE state = state_id;
```

  - a PL/SQL program cannot pass a cursor as a parameter to another program

Cursor Types and Variables

- Cursor type and its variable
  - can reference and pass a cursor variable as a parameter
  - two types
    - *strong* : including a **RETURN** clause that specifies a shape or set of attributes for the cursor type
    - *weak* : not including a shape specification

```
-- STRONG, SPECIFIC CURSOR TYPE
TYPE parts_type IS REF CURSOR RETURN sales.parts%ROWTYPE;
-- AND CORRESPODING CURSOR VARIABLES
parts_cursor1 parts_type;
parts_cursor2 parts_type;
…
TYPE cursor_type IS REF CURSOR; -- WEAK CURSOR TYPE
```

Assignment Statements(1/3)

- Example Scalar Variable Assignments

```
DECLARE
  emp_id INTEGER;
  another_integer_variable INTEGER := 0;
  part_description VARCHAR2(200);
BEGIN
  emp_id := 1;
  emp_id := another_integer_variable;
  part_description := 'Network Computer';
  …
```

Assignment Statements(2/3)

- Example Record Variable Assignments

```
DECLARE
  TYPE part_record IS RECORD (
    id INTEGER,
    unit_price NUMBER(10,2),
    description VARCHAR2(200)
  );
  current_part part_record;
  another_Part_Record_variable part_record;
BEGIN
  current_part.id := 1;
  current_part.description := ‘Network Computer’;
  current_part := another_Part_Record_variable;
  …
```

Assignment Statements(3/3)

- Example Nested Table or Varray Variable Assignments

```
DECLARE
  TYPE part_record IS RECORD (
    id INTEGER,
    unit_price NUMBER(10,2),
    description VARCHAR2(200)
  );
  TYPE parts_table IS TABLE OF part_record;
  current_parts_table parts_table;
BEGIN
  current_parts_table(1).id := 1;
  current_parts_table(1).description := ‘Network Computer’;
  current_parts_table(2) := current_parts_table(1);
  …
```

Nested Tables and Variable Arrays

- Comparison
  - nested tables
    - the size can increase or decrease dynamically
    - sparseness : can remove individual members of non-consecutive row in the table
  - variable arrays
    - a constant number of rows
    - densely space : must insert members into a varray using consecutive subscripts
- Initialization

```
TYPE parts_table IS TABLE OF sales.parts%ROWTYPE;
current_parts_table parts_table := parts_table (
 (1, 150.90, ‘Pentium 166 CPU’), NULL,
 (3, 500.00, ‘Network Computer’));
 …
```

a default constructor function

Collection Methods with Nested Tables and Varrays

- *EXISTS(x)* : TRUE if the x^{th} element in a nested table or varray exists. otherwise, FALSE
- *COUNT* : the number of current elements
- *LIMIT* : *for varrays*, the maximum number of elements that the collection can contain
- *FIRST/LAST* : the first/last member of the nested table or varray
- *PRIOR(x)/NEXT(x)* : the member prior/after to the xth member of the nested table or varray
- *EXTEND(x,y)* : appends x copies of the y^{th} element to a nested table or varray
- *TRIM(x)* : trim x elements from the end of a nested table or varray
- *DELETE(x,y)* : delete a nested table’s or varray’s $x\sim y^{th}$ elements

Some Example
Using Collection Methods

```
record_count := current_parts_table.COUNT;
current_parts_record := current_parts_table.FIRST;
current_parts_table.DELETE(3);
current_parts_table.DELETE(3,6); -- REMOVE 3~6TH ELEMENTS
current_parts_table.DELETE(6,3); -- DO NOTHING
current_parts_table.DELETE; -- REMOVE ALL THE ELEMENTS
current_parts_record :=
   current_parts_table.PRIOR(current_parts_table.FIRST);
   -- ASSIGN CURRENT_PARTS_RECORD TO NULL
current_parts_table.EXTEND(3,6);
   -- APPEND 3 copies of 6TH ELEMENTS
current_parts_table.EXTEND; -- APPEND 1 ELEMENT
current_parts_table.TRIM(3); -- REMOVE THE LAST 3 ELEMENTS

FOR i IN courses.FIRST..courses.LAST LOOP ...
```

Condition Control

```
-- BASIC IF STATEMENT
IF condition THEN
  statements;
END IF;

-- IF-ELSE STATEMENT
IF condition THEN
  statements 1;
ELSE
  statements 2;
END IF;
```

```
-- MORE COMPLEX IF-ELSIF-ELSE
  STATEMENT
IF condition 1 THEN
  statements 1;
ELSIF condition 2 THEN
  GOTO section_1;
ELSIF condition 3 THEN
  statement 3;
ELSIF condition 4 THEN
  statement 4;
END IF;
…
<<section_1>>
…
```

Iterative Control

```
-- BASIC LOOP
LOOP
  statement 1;
  statement 2;
  …
  EXIT WHEN condition;
END LOOP;

-- WHILE LOOP
WHILE condition LOOP
  statement 1;
  statement 2;
  …
END LOOP;
```

```
-- FOR LOOP WITH NESTED-LOOP
<<outer_loop>> -- loop label
FOR x IN y..z LOOP
  outer_statement 1;
  <<inner_loop>>
  LOOP
    inner_statement 1;
    inner_statement 2;
    EXIT outer_loop WHEN condition1;
    EXIT inner_loop WHEN condition2;
  END LOOP inner_loop;
  …
END LOOP outer_loop;
…
```

Database Interaction

- Standard DML
  - PL/SQL programs can use any SQL DML statement
- SELECT INTO

```
DECLARE
  current_part sales.parts%ROWTYPE;
BEGIN
  SELECT * INTO current_part
    FROM sales.parts
    WHERE id = 6;
```

  - if the result set contains more than one row, return an error
    - a PL/SQL program must use a cursor

Working With Cursors

- Three steps
  - open the cursor ➔ fetch the rows ➔ close the cursor

```
DECLARE
  CURSOR parts_cursor IS SELECT * FROM sales.parts;
  current_part sales.parts%ROWTYPE;
BEGIN
  OPEN parts_cursor; -- OPEN the cursor
  LOOP
    FETCH pars_cursor INTO current_part; -- FETCH rows
    …
  END LOOP;
  CLOSE parts_cursor; -- CLOSE the cursor
  …
```

Cursor FOR Loop (1/3)

- function
  - automatically declare a variable or record capable of receiving the rows in the cursor, open the cursor, fetch rows, and close the cursor when the last fetch operation

```
DECLARE
  current_part sales.parts%ROWTYPE;
  CURSOR parts_cursor IS
    SELECT * FROM sales.parts;
BEGIN
  FOR current_part IN parts_cursor LOOP
    …
  END LOOP;
  …
```

Cursor FOR Loop (2/3)

- With cursor parameters

```
DECLARE
  CURSOR customers_cursor (state_id CHAR) IS
    SELECT * FROM sales.customers
    WHERE state = state_id;
BEGIN
  FOR current_customer IN customers_cursor(‘CA’) LOOP
    …
  END LOOP;
  …
```

- Explicit cursor attributes
  - %ISOPEN, %FOUND, %NOTFOUND, %ROWCOUNT(the number of rows fetched so far)

```
WHILE customers_cursor%FOUND LOOP … END LOOP;
```

Cursor FOR Loop (3/3)

- CURRENT OF in UPDATE/DELETE statements

```
BEGIN
  FOR current_customer IN customers_cursor(‘CA’) LOOP
    IF … THEN
      DELETE FROM sales.customers
        WHERE CURRENT OF customers_cursor;
    END IF;
  END LOOP;
  …
```

- Using cursor variables
  - cannot use a cursor FOR loop construct
  - open a cursor using an OPEN FOR statement

Working With Cursor Variables

```
DECLARE
  TYPE cursor_type IS REF CURSOR;
  customers_cursorv cursor_type;
  current_customer sales.customers%ROWTYPE;
BEGIN
  OPEN customers_cursorv FOR
    SELECT id, last_name, first_name, phone
    FROM sales.customers;
  WHILE customers_cursorv%FOUND LOOP
    FETCH customers_cursorv INTO current_customer;
    IF … THEN
      …
    END IF;
    …
  END LOOP;
  CLOSE customers_cursorv;
  …
```

Dynamic SQL

- Static vs. Dynamic
  - static SQL
    - – bind all SQL at compile-time
    - – cannot execute SQL DDL statements
    - – maximum performance, inflexible
  - dynamic SQL
    - – create and bind SQLs at run-time
    - – flexible, poor performance
- Three ways to perform dynamic SQL
  1. DBMS_SQL package : required for performing dynamic SQL
  2. Native dynamic SQL :
     EXECUTE IMMEDIATE statement(single-row retrieval),
     OPEN FOR statement(multi-row retrieval)

DBMS_SQL Package

```
CREATE OR REPLACE PROCEDURE utilities.drop_table (
  schema_name IN OUT VARCHAR2,
  table_name IN OUT VARCHAR2
) IS
  cursor_id INTEGER;
  return_value INTEGER;
  command_string VARCHAR2(250);
BEGIN
  command_string := 'DROP TABLE ' || schema_name || '.' ||
    table_name;
  cursor_id := dbms_sql.open_cursor;
  dbms_sql.parse(cursor_id, command_string, dbms_sql.v7);
  return_value := dbms_sql.execute(cursor_id);
  dbms_sql.close_cursor(cursor_id);
END drop_table;
```

EXECUTE IMMEDIATE Statement

```
DECLARE
   sql_stmt VARCHAR2(100);
   my_deptno NUMBER(2) := 50;
   my_dname VARCHAR2(15) := 'PERSONNEL';
   my_loc VARCHAR2(15) := 'DALLAS';
   emp_rec emp%ROWTYPE;
BEGIN
   sql_stmt := 'INSERT INTO dept VALUES (:1, :2, :3)';
   EXECUTE IMMEDIATE sql_stmt USING my_deptno, my_dname, my_loc;
   sql_stmt := 'SELECT * FROM emp WHERE empno = :id';
   EXECUTE IMMEDIATE sql_stmt INTO emp_rec USING 7788;
   EXECUTE IMMEDIATE 'DELETE FROM dept WHERE deptno = :n' USING
     my_deptno;
   …
   EXECUTE IMMEDIATE 'CREATE TABLE bonus (id NUMBER, amt NUMBER)';
   sql_stmt := 'ALTER SESSION SET SQL_TRACE TRUE';
   EXECUTE IMMEDIATE sql_stmt;
END;
```

OPEN FOR Statement

```
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;
    sql_string VARCHAR2(50);
BEGIN
    sql_string := 'SELECT ename, sal FROM emp WHERE sal > :s'
    OPEN emp_cv FOR sql_string USING my_sal; -- open cursor variable
    LOOP
        FETCH emp_cv INTO my_ename, my_sal; -- fetch next row
        EXIT WHEN emp_cv%NOTFOUND;
        …
    END LOOP;
    CLOSE emp_cv; -- close cursor variable
    …
```

Exception Handling (1/3)

- Error Handling
  - PL/SQL program *raises* a named exception when it detects an error
  - passing control to an associated exception handler routine
- Exception
  - a named error condition
  - almost 20 predefined exceptions
    - NO_DATA_FOUND, TOO_MANY_ROWS in a SELECT statement
    - DUP_VAL_ON_INDEX in an INSERT or UPDATE statement
    - ZERO_DIVIDE
  - user-defined exceptions in the declaration section
    - a program must perform explicit checks for a user-defined exception

Exception Handling (2/3)

```
DECLARE
   invalid_part EXCEPTION;
   insufficient_privileges EXCEPTION;
   PRAGMA EXCEPTION_INIT (insufficient_privileges, -1031);
   err_num INTEGER;
   err_msg VARCHAR2(2000);
   part_num INTEGER;
BEGIN
   SELECT … INTO … FROM …;
   UPDATE sales.parts
     SET unit_price = 20.00 WHERE id = 6;
   IF SQL%NOTFOUND THEN
     RAISE invalid_part;
   END IF;
EXCEPTION
   WHEN no_data_found THEN
     raise_application_error(-20001, 'No rows found');
```

compiler directive : associate an exception name with an Oracle error number

Exception Handling (3/3)

```
   WHEN too_many_rows THEN
     raise_application_error (-20002, 'Too many rows found');
   WHEN invalid_part THEN
     raise_application_error (-20003, 'Invalid Part ID');
   WHEN insufficient_privileges THEN
     raise_application_error (-20004,
          'Insufficient privileges to update table');
   WHEN OTHERS THEN
     err_num := SQLCODE;
     err_msg := SUBSTR(SQLERRM, 1, 100);
     raise_application_error (-20000, err_num || ' ' || err_msg);
   …
```

- all user-defined error messages must be in the range –20000 to –20999
- **WHEN OTHERS THEN** : a generic exception without a specific error handler

Types of PL/SQL Programs

- Anonymous PL/SQL blocks, procedures, functions, and packages
- Anonymous PL/SQL block
  - a PL/SQL block that appears within an application
  - no name, no storage in a database
  - simply sending the block of code to the database server for processing at runtime
  - beginning with DECLARE and ending with END

Stored Subprograms (1/3)

- Subprogram
  - a named PL/SQL program that can take parameters and be called by an application
  - can store compiled bits of application logic inside an Oracle database using stored subprograms, as schema objects
    - – stored procedures
    - – stored functions : returning a value
  - the commands CREATE PROCEDURE or CREATE FUNCTION
- Parameters
  - three modes : IN, OUT, IN OUT
- Stored functions
  - must have one or more RETURN statements

Stored Subprograms (2/3)

```
CREATE OR REPLACE FUNCTION sales.get_customer_id (
   last IN VARCHAR2, first IN VARCHAR2
)
RETURN INTEGER IS
   cust_id INTEGER;
BEGIN
   SELECT id INTO cust_id
     FROM sales.customers
     WHERE last_name = last AND fist_name = first;
   RETURN cust_id;
EXCEPTION
   WHEN OTHERS THEN
     RETURN NULL;
END get_customer_id;
```

Stored Subprograms (3/3)

- Calling procedures and functions
  - call a procedure by reference with all parameters
  - call a function by reference in an assignment statement or a WHEN clause

```
DECLARE
  cur_cust_id INTEGER;
  cur_cust_last VARCHAR2(100);
  cur_cust_first VARCHAR2(100);
BEGIN
  …
  cur_cust_id :=
    sales.get_customer_id(cur_cust_last, cur_cust_first);
  …
  DELETE FROM sales.orders
    WHERE cust_id = sales.get_customer_id(‘Ellison’, ‘Lawrence’);
```

Packages (1/5)

- Definition
  - a group of procedures, functions, and other PL/SQL constructs, all stored together in a database as a unit
- Structure
  - specification
    - the interface to the package
    - declaration of all package variables, constants, cursors, procedures, functions, and other exported constructs
    - everything in a specification is *public*
  - body
    - definition of all public procedures and functions
    - package constructs are *private*
  - all declared variables, constants, and cursors are *global*

Packages (2/5)

```
CREATE OR REPLACE PACKAGE sales.part_mgmt IS
-- GLOBAL TYPES AND VARIABLES
   TYPE parts_type IS REF CURSOR RETURN sales.parts%ROWTYPE;
   current_part sales.parts%ROWTYPE;
-- PROCEDURES AND FUNCTIONS
   PROCEDURE insert_part (part_record sales.parts%ROWTYPE);
   PROCEDURE update_part_unitprice (part_id IN INTEGER,
                                    new_price IN NUMBER);
   PROCEDURE update_part_description (part_id IN INTEGER,
                                      new_desc IN NUMBER);
   PROCEDURE delete_part (part_id IN INTEGER);
  FUNCTION get_part_id (part_desc IN VARCHAR2) RETURN INTEGER;
END part_mgmt;
```

Packages (3/5)

```
CREATE OR REPLACE PACKAGE BODY sales.part_mgmt IS
   -- some private global constructs
   unit_price INTEGER;
   …
   PROCEDURE insert_part (part_record sales.parts%ROWTYPE) IS
     dup_primary_key EXCEPTION;
     PRAGMA EXCEPTION_INIT (dup_primary_key, -1);
   BEGIN
     INSERT INTO sales.parts
      VALUES (part_record.id, part_record.unit_price,
                                    part_record.description);
   EXCEPTION
     WHEN dup_primary_key THEN
      raise_application_error(-20001, ‘Duplicate part ID’);
     WHEN OTHERS THEN
      raise_application_error(-20000, ‘Undefined exception’);
   END insert_part;
   … other package procedure and function definitions …
END part_mgmt;
```

Packages (4/5)

- Using package objects

```
DECLARE
BEGIN
-- THIS STATEMENT INITIALIZES A GLOBAL PACKAGE VARIABLE
  SELECT * INTO sales.part_mgmt.current_part
    FROM sales.parts
    WHERE id = 3;
-- THIS STATEMENT CALLS THE INSERT_PART PACKAGED PROCEDURE
  sales.part_mgmt.insert_part(3,500.00,‘Network Computer’);
  …
```

Packages (5/5)

- DBMS utility packages
  - DBMS_ALERT : allowing applications to name and signal alert conditions without polling
  - DBMS_AQ, DBMS_AQADM : queuing the execution of transactions and administering queuing mechanisms
  - DBMS_DDL, DBMS_UTILITY : allowing applications to access some of DDL statements
  - DBMS_DESCRIBE : API description for stored subprograms
  - DBMS_ROWID : allowing applications to easily interpret a base-64 character external ROWID
  - DBMS_SQL : performing dynamic SQLs
  - UTL_FILE : reading and writing text files to the server's file system
  - DBMS_JOB, DBMS_LOB, DBMS_LOCK, DBMS_PIPE, DBMS_SESSION, DBMS_ROWID, DBMS_TRANSACTION …

Triggers (1/3)

- Database trigger
  - a stored procedure that you associate with a table
  - Event-Condition-Action rule
- Type of triggers
  - statement trigger : firing the trigger only once, no matter how many rows the trigger statement affects
  - row trigger : firing once for each row that the trigger statement affects
- Components
  - predicates : INSERTING, UPDATING, DELETING
  - new and old values of the current row
    - :new, :old

Triggers (2/3)

```
CREATE OR REPLACE TRIGGER sales.parts_log
AFTER INSERT OR UPDATE OR DELETE ON sales.parts
DECLARE
   stmt_type CHAR(1);
BEGIN
   IF INSERTING THEN
     stmt_type := ‘I’;
   ELSIF UPDATING THEN
     stmt_type := ‘U’;
   ELSE
     stmt_type := ‘D’;
   END IF;
   INSERT INTO sales.part_change_log
     VALUES (stmt_type, USER);
END parts_log;
```

Triggers (3/3)

```
CREATE OR REPLACE TRIGGER sales.parts_log
AFTER INSERT OR UPDATE OR DELETE ON sales.parts
FOR EACH ROW
DECLARE
   stmt_type CHAR(1);
BEGIN
   IF INSERTING THEN
     stmt_type := ‘I’;
   ELSIF UPDATING THEN
     stmt_type := ‘U’;
   ELSE
     stmt_type := ‘D’;
   END IF;
   INSERT INTO sales.part_change_log
    VALUES (:new.id, :old.id, :new.unit_price, :old.unit_price,
     :new.description, :old.description, stmt_type, USER
    );
END parts_log;
```

External Procedures (1/2)

- Features
  - a PL/SQL program can make use of external procedures within external shared program libraries
  - can take full advantage of existing code without having to rewrite it as PL/SQL
  - Oracle safely executes an external procedure in its own address space on the server
- Usage
  1. write or make available the compiled shared program library
  2. use the SQL command CREATE LIBRARY to declare a name for the shared program library
  3. write simple PL/SQL procedures or functions to call external procedures and functions

External Procedures (2/2)

```
CREATE LIBRARY external.odbc as ‘c:\windows\system\odbc.dll’;

CREATE OR REPLACE FUNCTION external.sql_exec_direct (
-- EXECUTE ANY SQL STATEMENT USING ODBC
   sql_handle BINARY_INTEGER;
   sql_statement VARCHAR2(2000),
   sql_length INTEGER )
RETURN VARCHAR2 AS EXTERNAL
   LIBRARY external.odbc
   NAME SQLExecDirect
   LANGUAGE C;
```

```
-- CALLING PROCEDURE
DECLARE
   return_code VARCHAR2(2000);
   stmt VARCHAR2(2000) := ‘DELETE FROM access.customers’;
BEGIN
   return_code := external.sql_exec_direct(1, stmt, LENGTH(stmt));
   …
```