The PostgreSQL PL/pgSQL control statements

Declaration statements

- A **declaration statement** allows the developer to define variables and constants.
- The general syntax of a variable declaration is as follows:

```
name [ CONSTANT ] type [ COLLATE collation_name ] [ NOT NULL ] [ { DEFAULT | := | = } expression ];
```

- The PostgreSQL PL/pgSQL function body is composed of nested blocks, with an optional declaration section and a label.
- Variables are declared in the `DECLARE` section of the block, as follows:

```
[ <<label>> ]
[ DECLARE
    declarations ]
BEGIN
    statements
END [ label ];
```

- To understand the function block, let's take a look at the following code, which defines the `factorial` function in a recursive manner:

```
car_portal=# CREATE OR REPLACE FUNCTION factorial(INTEGER ) RETURNS INTEGER AS $$
car_portal$# BEGIN
car_portal$#         IF $1 IS NULL OR $1 < 0 THEN RAISE NOTICE 'Invalid Number';
car_portal$#                 RETURN NULL;
car_portal$#         ELSIF $1 = 1 THEN
car_portal$#                 RETURN 1;
car_portal$#         ELSE
car_portal$#                 RETURN factorial($1 - 1) * $1;
car_portal$# END IF;
car_portal$# END;
car_portal$# $$ LANGUAGE 'plpgsql';
CREATE FUNCTION
car_portal=#
car_portal=# select factorial(5);
 factorial
-----------
       120
(1 row)
```

- The block defines the variable scope; in our example, the scope of the argument variable, `$1`, is the whole function.
- Also, as shown in the example, there is no declaration section.
- To understand the scope between different code blocks, let's write the factorial function in a slightly different manner, as follows:

```
car_portal=# CREATE OR REPLACE FUNCTION factorial(INTEGER ) RETURNS INTEGER AS $$
car_portal$#         DECLARE
car_portal$#                 fact ALIAS FOR $1;
car_portal$#         BEGIN
car_portal$#         IF fact IS NULL OR fact < 0 THEN
car_portal$#                 RAISE NOTICE 'Invalid Number';
car_portal$#                 RETURN NULL;
car_portal$#         ELSIF fact = 1 THEN
car_portal$#                 RETURN 1;
car_portal$#         END IF;
car_portal$#         DECLARE
car_portal$#                 result INT;
car_portal$#         BEGIN
car_portal$#                 result = factorial(fact - 1) * fact;
car_portal$#                 RETURN result;
car_portal$#         END;
car_portal$# END;
car_portal$# $$ LANGUAGE 'plpgsql';
CREATE FUNCTION
car_portal=#
car_portal=# select factorial(5);
 factorial
-----------
       120
(1 row)
```

- The preceding function is composed of two blocks; the `fact` variable is an alias for the first argument.
- In the sub-block, the `result` variable is declared with a type integer.
- Since the `fact` variable is defined in the upper block, it can also be used in the sub-block.
- The `result` variable can only be used in the sub-block.

Assignment statements

- An **assignment statement** is used to assign a value to a variable.
- Constants are assigned values at the time of declaration.
- The variable can be assigned an atomic value or a complex value, such as a record.
- Also, it can be assigned a literal value or a result of query execution.

- The assignment := and =, operators are used to assign an expression to a variable, as follows:
  ```
  variable { := | = } expression;
  ```
- For the variable names, you should choose names that do not conflict with the column names.
- This is important when writing parameterized SQL statements.
- The = operator is used in SQL for equality comparisons; it is preferable to use the := operator, in order to reduce confusion.

- In certain contexts, the operators = and := cannot be used interchangeably; it is important to pick the correct assignment operator for the following cases:
  - When assigning a default value, you should use the = operator, as indicated in the documentation at http://www.postgresql.org/docs/current/interactive/sql-createfunction.html
  - For named notations in a function call, you should use the := operator

- The following example shows a case in which we cannot use = and := interchangeably:

```
car_portal=# CREATE OR REPLACE FUNCTION cast_numeric_to_int (numeric_value numeric, round boolean = TRUE /*correct use of "
=". Using ":=" will raise a syntax error */)
car_portal-# RETURNS INT AS
car_portal-# $$
car_portal$# BEGIN
car_portal$#   RETURN (CASE WHEN round = TRUE THEN CAST (numeric_value AS INTEGER)
car_portal$#                WHEN numeric_value >= 0 THEN CAST (numeric_value -.5 AS INTEGER)
car_portal$#                WHEN numeric_value < 0 THEN CAST (numeric_value +.5 AS INTEGER)
car_portal$#                ELSE NULL
car_portal$#           END);
car_portal$# END;
car_portal$# $$ LANGUAGE plpgsql;
CREATE FUNCTION
car_portal=#
```

- To show how to call the `cast_numeric_to_int` function, let's execute the following code:

```
car_portal=# SELECT cast_numeric_to_int(2.3, round := true);
 cast_numeric_to_int
---------------------
                   2
(1 row)

car_portal=# SELECT cast_numeric_to_int(2.3, round = true);
ERROR:  column "round" does not exist
LINE 1: SELECT cast_numeric_to_int(2.3, round = true);
                                        ^
```

- The assignment expression can be a single atomic value, such as `pi = 3.14`, or it can be a row, as shown in the following code snippet:

```
car_portal=# DO $$
car_portal$#   DECLARE
car_portal$#     test record;
car_portal$#   BEGIN
car_portal$#     test = ROW (1,'hello', 3.14);
car_portal$#     RAISE notice '%', test;
car_portal$#   END;
car_portal$# $$ LANGUAGE plpgsql;
NOTICE:  (1,hello,3.14)
DO
car_portal=#
car_portal=# DO $$
car_portal$#   DECLARE
car_portal$#     number_of_accounts INT :=0;
car_portal$#   BEGIN
car_portal$#     number_of_accounts := (SELECT COUNT(*) FROM car_portal_app.account)::INT;
car_portal$#     RAISE NOTICE 'number_of accounts: %', number_of_accounts;
car_portal$#   END;
car_portal$# $$ LANGUAGE plpgsql;
NOTICE:  number_of accounts: 481
DO
car_portal=#
```

- There are other techniques for assigning values to variables from a query that returns a single row:
  - `SELECT select_expressions INTO [STRICT] targets FROM ...;`
  - `INSERT ... RETURNING expressions INTO [STRICT] targets;`
  - `UPDATE ... RETURNING expressions INTO [STRICT] targets;`
  - `DELETE ... RETURNING expressions INTO [STRICT] targets;`

- Often, expressions are column names, while targets are variable names.
- In the case of `SELECT INTO`, the target can be of the `record` type.
- The `INSERT … RETURNING` query is often used to return the default value of a certain column; this can be used to define the ID of a primary key, using the `SERIAL` and `BIGSERIAL` data types.

```
car_portal=# CREATE TABLE test (
car_portal(#   id SERIAL PRIMARY KEY,
car_portal(#   name TEXT NOT NULL
car_portal(# );
CREATE TABLE
car_portal=# DO $$
car_portal$#   DECLARE
car_portal$#     auto_generated_id INT;
car_portal$#   BEGIN
car_portal$#     INSERT INTO test(name) VALUES ('Hello World') RETURNING id INTO auto_generated_id;
car_portal$#     RAISE NOTICE 'The primary key is: %', auto_generated_id;
car_portal$#   END
car_portal$# $$;
NOTICE:  The primary key is: 1
DO
car_portal=#
```

- You can get the default value when inserting a row in plain SQL by using CTE, as follows:

```
WITH get_id AS (
   INSERT INTO test(name) VALUES ('Hello World’)
   RETURNING id
)
SELECT * FROM get_id;
```

- Finally, you can use qualified names to perform assignments; in trigger functions, you can use `NEW` and `OLD` to manipulate the values of these records.

Conditional statements

- PostgreSQL supports the `IF` and `CASE` statements, which allow for execution based on a certain condition.
- PostgreSQL supports the `IF` statement construct, as follows:
  - `IF ... THEN ... END IF`
  - `IF ... THEN ... ELSE ... END IF`
  - `IF ... THEN ... ELSIF ... THEN ... ELSE ... END IF`
- The `CASE` statement comes in two forms, as follows:
  - `CASE ... WHEN ... THEN ... ELSE ... END CASE`
  - `CASE WHEN ... THEN ... ELSE ... END CASE`

- To understand the `IF` statement, let's suppose that we would like to convert the `advertisement` rank into text, as follows:

```
car_portal=# CREATE OR REPLACE FUNCTION cast_rank_to_text (rank int) RETURNS TEXT AS $$
car_portal$# DECLARE
car_portal$#   rank ALIAS FOR $1;
car_portal$#   rank_result TEXT;
car_portal$# BEGIN
car_portal$#   IF rank = 5 THEN rank_result = 'Excellent';
car_portal$#   ELSIF rank = 4 THEN rank_result = 'Very Good';
car_portal$#   ELSIF rank = 3 THEN rank_result = 'Good';
car_portal$#   ELSIF rank = 2 THEN rank_result ='Fair';
car_portal$#   ELSIF rank = 1 THEN rank_result ='Poor';
car_portal$#   ELSE rank_result ='No such rank';
car_portal$#   END IF;
car_portal$#   RETURN rank_result;
car_portal$# END;
car_portal$# $$ Language plpgsql;
CREATE FUNCTION
car_portal=#
```

- To test the function, run the following command:

```
car_portal=# SELECT n,cast_rank_to_text(n) FROM generate_series(1,6) as foo(n);
 n | cast_rank_to_text
---+-------------------
 1 | Poor
 2 | Fair
 3 | Good
 4 | Very Good
 5 | Excellent
 6 | No such rank
(6 rows)
```

- The following code snippet implements the preceding function, using the `CASE` statement:

```
car_portal=# CREATE OR REPLACE FUNCTION cast_rank_to_text (rank int) RETURNS TEXT AS $$
car_portal$# DECLARE
car_portal$#   rank ALIAS FOR $1;
car_portal$#   rank_result TEXT;
car_portal$# BEGIN
car_portal$#   CASE rank
car_portal$#     WHEN 5 THEN rank_result = 'Excellent';
car_portal$#     WHEN 4 THEN rank_result = 'Very Good';
car_portal$#     WHEN 3 THEN rank_result = 'Good';
car_portal$#     WHEN 2 THEN rank_result ='Fair';
car_portal$#     WHEN 1 THEN rank_result ='Poor';
car_portal$#     ELSE rank_result ='No such rank';
car_portal$#   END CASE;
car_portal$#   RETURN rank_result;
car_portal$# END;$$ Language plpgsql;
CREATE FUNCTION
car_portal=# SELECT n,cast_rank_to_text(n) FROM generate_series(1,6) as foo(n);
 n | cast_rank_to_text
---+-------------------
 1 | Poor
 2 | Fair
 3 | Good
 4 | Very Good
 5 | Excellent
 6 | No such rank
(6 rows)
```

- In the previous form of `CASE`, we cannot use it to match `NULL` values, since `NULL` equality is `NULL`.
- To overcome this limitation, you should specify the matching condition explicitly, using the second form of the `CASE` statement, as follows:

```
car_portal=# CREATE OR REPLACE FUNCTION cast_rank_to_text (rank int) RETURNS TEXT AS $$
car_portal$# DECLARE
car_portal$#   rank ALIAS FOR $1;
car_portal$#   rank_result TEXT;
car_portal$# BEGIN
car_portal$#   CASE
car_portal$#     WHEN rank=5 THEN rank_result = 'Excellent';
car_portal$#     WHEN rank=4 THEN rank_result = 'Very Good';
car_portal$#     WHEN rank=3 THEN rank_result = 'Good';
car_portal$#     WHEN rank=2 THEN rank_result ='Fair';
car_portal$#     WHEN rank=1 THEN rank_result ='Poor';
car_portal$#     WHEN rank IS NULL THEN RAISE EXCEPTION 'Rank should be not NULL';
car_portal$#     ELSE rank_result ='No such rank';
car_portal$#   END CASE;
car_portal$#   RETURN rank_result;
car_portal$# END;
car_portal$# $$ Language plpgsql;
CREATE FUNCTION
car_portal=#
```

- To test the function, use the following command:

```
car_portal=# SELECT cast_rank_to_text(null);
ERROR:  Rank should be not NULL
CONTEXT:  PL/pgSQL function cast_rank_to_text(integer) line 12 at RAISE
car_portal=#
```

- Finally, the `CASE` statement raises an exception if no branch is matched and the `ELSE` branch is not specified, as follows:

```
car_portal=# DO $$
car_portal$# DECLARE
car_portal$#   i int := 0;
car_portal$# BEGIN
car_portal$#   case WHEN i=1 then
car_portal$#     RAISE NOTICE 'i is one';
car_portal$#   END CASE;
car_portal$# END;
car_portal$# $$ LANGUAGE plpgsql;
ERROR:  case not found
HINT:  CASE statement is missing ELSE part.
CONTEXT:  PL/pgSQL function inline_code_block line 5 at CASE
car_portal=#
```

Iterations

Loop statements

- The basic `LOOP` statement has the following structure:

```
[ <<label>> ]
LOOP
    statements
END LOOP [ label ];
```

- To understand the `LOOP` statement, let's rewrite the `factorial` function, as follows:

```
car_portal=# DROP FUNCTION IF EXISTS factorial (int);
DROP FUNCTION
car_portal=# CREATE OR REPLACE FUNCTION factorial (fact int) RETURNS BIGINT AS $$
car_portal$# DECLARE
car_portal$#   result bigint = 1;
car_portal$# BEGIN
car_portal$#   IF fact = 1 THEN RETURN 1;
car_portal$#   ELSIF fact IS NULL or fact < 1 THEN RAISE EXCEPTION 'Provide a positive integer';
car_portal$#   ELSE
car_portal$#     LOOP
car_portal$#       result = result*fact;
car_portal$#       fact = fact-1;
car_portal$#       EXIT WHEN fact = 1;
car_portal$#     END Loop;
car_portal$#   END IF;
car_portal$#   RETURN result;
car_portal$# END; $$ LANGUAGE plpgsql;
CREATE FUNCTION
car_portal=#
```

```
car_portal=# select factorial(5);
 factorial
-----------
       120
(1 row)
```

The WHILE loop statement

- The `WHILE` statement keeps executing a block of statements while a particular condition is met.
- Its syntax is as follows:

```
[ <<label>> ]
WHILE boolean-expression LOOP
    statements
END LOOP [ label ];
```

- The following example uses the `while` loop to print the days of the current month:

```
car_portal=# DO $$
car_portal$# DECLARE
car_portal$#   first_day_in_month date := date_trunc('month', current_date)::date;
car_portal$#   last_day_in_month date := (date_trunc('month', current_date)+ INTERVAL '1 MONTH - 1 day')::date;
car_portal$#   counter date = first_day_in_month;
car_portal$# BEGIN
car_portal$#   WHILE (counter <= last_day_in_month) LOOP
car_portal$#     RAISE notice '%', counter;
car_portal$#     counter := counter + interval '1 day';
car_portal$#   END LOOP;
car_portal$# END;
car_portal$# $$ LANGUAGE plpgsql;
NOTICE:  2019-10-01
NOTICE:  2019-10-02
NOTICE:  2019-10-03
NOTICE:  2019-10-04
NOTICE:  2019-10-05
NOTICE:  2019-10-06
NOTICE:  2019-10-07
NOTICE:  2019-10-08
NOTICE:  2019-10-09
NOTICE:  2019-10-10
NOTICE:  2019-10-11
NOTICE:  2019-10-12
NOTICE:  2019-10-13
NOTICE:  2019-10-14
NOTICE:  2019-10-15
```

The FOR loop statement

- PL/pgSQL provides two forms of the `FOR` statement, and they are used to execute the following:
  - Iterate through the rows returned by an SQL query
  - Iterate through a range of integer values

- The syntax of the FOR loop statement is as follows:

```
[ <<label>> ]
FOR name IN [ REVERSE ] expression1 .. expression2 [ BY expression ]
LOOP
    statementsEND
LOOP [ label ];
```

- The following example shows a `FOR` loop iterating over a negative range of numbers in a reverse order.

```
car_portal=# DO $$
car_portal$# BEGIN
car_portal$#   FOR j IN REVERSE -1 .. -10 BY 2 LOOP
car_portal$#     Raise notice '%', j;
car_portal$#   END LOOP;
car_portal$# END; $$ LANGUAGE plpgsql;
NOTICE:  -1
NOTICE:  -3
NOTICE:  -5
NOTICE:  -7
NOTICE:  -9
DO
car_portal=#
```

- To iterate through the results of a set query, the syntax is different, as follows:

```
[ <<label>> ]
FOR target IN query LOOP
    statements
END LOOP [ label ];
```

- The following example shows all of the database names:

```
car_portal=# DO $$
car_portal$# DECLARE
car_portal$#   database RECORD;
car_portal$# BEGIN
car_portal$#   FOR database IN SELECT * FROM pg_database LOOP
car_portal$#     RAISE notice '%', database.datname;
car_portal$#   END LOOP;
car_portal$# END; $$;
NOTICE:  postgres
NOTICE:  car_portal
NOTICE:  template1
NOTICE:  template0
DO
car_portal=#
```

Returning from the function

Returning voids

- A `void` type is used in a function to perform some side effects, such as logging; the built-in function, `pg_sleep`, is used to delay the execution of a server process, in seconds, as follows:

```
car_portal=# \df pg_sleep
                              List of functions
   Schema   |   Name   | Result data type | Argument data types | Type
------------+----------+------------------+---------------------+------
 pg_catalog | pg_sleep | void             | double precision    | func
(1 row)
```

Returning a single row

- The RETURN type can be a base, composite, table, pseudo, or domain data type.
- The following function returns a `JSON` representation of a certain account:

```
car_portal=# CREATE OR REPLACE FUNCTION car_portal_app.get_account_in_json (account_id INT) RETURNS JSON AS $$
car_portal$#   SELECT row_to_json(account) FROM car_portal_app.account WHERE account_id = $1;
car_portal$# $$ LANGUAGE SQL;
CREATE FUNCTION
car_portal=# CREATE OR REPLACE FUNCTION car_portal_app.get_account_in_json1 (acc_id INT) RETURNS JSON AS $$
car_portal$# BEGIN
car_portal$#   RETURN (SELECT row_to_json(account) FROM car_portal_app.account WHERE account_id = acc_id);
car_portal$# END;
car_portal$# $$ LANGUAGE plpgsql;
CREATE FUNCTION
```

```
car_portal=# select car_portal_app.get_account_in_json(1);
```

```
                                                get_account_in_json

-------------------------------------------------------------------------------------------------------------------------------
-------
 {"account_id":1,"first_name":"James","last_name":"Butt","email":"jbutt@gmail.com","password":"1b9ef408e82e38346e6ebebf2dcc
5ece"}
(1 row)
```

```
car_portal=# select car_portal_app.get_account_in_json1(1);
```

```
                                                get_account_in_json1

-------------------------------------------------------------------------------------------------------------------------------
-------
 {"account_id":1,"first_name":"James","last_name":"Butt","email":"jbutt@gmail.com","password":"1b9ef408e82e38346e6ebebf2dcc
5ece"}
(1 row)
```

Returning multiple rows

- **Set returning functions** (**SRFs**) can be used to return a set of rows.
- The row type can either be a base type, such as an integer, composite, table type, pseudo type, or domain type.
- To return a set from a function, the `SETOF` keyword is used to mark the function as an SRF, as follows:

```
car_portal=# CREATE OR REPLACE FUNCTION car_portal_app.car_model(model_name TEXT) RETURNS SETOF car_portal_app.car_model AS
 $$
car_portal$#   SELECT car_model_id, make, model FROM car_portal_app.car_model WHERE model = model_name;
car_portal$# $$ LANGUAGE SQL;
CREATE FUNCTION
car_portal=# CREATE OR REPLACE FUNCTION car_portal_app.car_model1(model_name TEXT) RETURNS SETOF car_portal_app.car_model A
S $$
car_portal$# BEGIN
car_portal$#   RETURN QUERY SELECT car_model_id, make, model FROM car_portal_app.car_model WHERE model = model_name;
car_portal$# END;
car_portal$# $$ LANGUAGE plpgsql;
CREATE FUNCTION
```

- To test the previous functions, let's run them.
- Note the caching effect of `plpgsql` on the performance:

```
car_portal=# \timing
Timing is on.
car_portal=# SELECT * FROM car_portal_app.car_model('A1');
 car_model_id | make | model
--------------+------+-------
            1 | Audi | A1
(1 row)

Time: 0.911 ms
car_portal=# SELECT * FROM car_portal_app.car_model1('A1');
 car_model_id | make | model
--------------+------+-------
            1 | Audi | A1
(1 row)

Time: 0.807 ms
```

- If the return type is not defined, you can do the following:
  - Define a new data type and use it
  - Use a return table
  - Use output parameters and record the data type

- Let's suppose that we would only like to return `car_model_id` and `make`; as in this case, we do not have a data type defined.
- The preceding function could be written as follows:

```
car_portal=# CREATE OR REPLACE FUNCTION car_portal_app.car_model2(model_name TEXT) RETURNS TABLE (car_model_id INT , make T
EXT) AS $$
car_portal$#   SELECT car_model_id, make FROM car_portal_app.car_model WHERE model = model_name;
car_portal$# $$ LANGUAGE SQL;
CREATE FUNCTION
car_portal=# CREATE OR REPLACE FUNCTION car_portal_app.car_model3(model_name TEXT) RETURNS TABLE (car_model_id INT , make T
EXT) AS $$
car_portal$# BEGIN
car_portal$#         RETURN QUERY SELECT car_model_id, make FROM car_portal_app.car_model WHERE model = model_name;
car_portal$# END;
car_portal$# $$ LANGUAGE plpgsql;
CREATE FUNCTION
```

- To test the `car_model2` and `car_model3` functions, let's run the following code:

```
car_portal=# SELECT * FROM car_portal_app.car_model2('A1');
 car_model_id | make
--------------+------
            1 | Audi
(1 row)

car_portal=# SELECT * FROM car_portal_app.car_model3('A1');
ERROR:  column reference "car_model_id" is ambiguous
LINE 1: SELECT car_model_id, make FROM car_portal_app.car_model WHER...
               ^
DETAIL:  It could refer to either a PL/pgSQL variable or a table column.
QUERY:  SELECT car_model_id, make FROM car_portal_app.car_model WHERE model = model_name
CONTEXT:  PL/pgSQL function car_portal_app.car_model3(text) line 3 at RETURN QUERY
car_portal=#
```

- Note that, in the preceding function, an error is raised, because `plpgsql` was confusing the column name with the table definition.
- The reason is that the return table is shorthand for writing `OUTPUT` parameters.
- To fix this, we need to rename the attribute names, as follows:

```
car_portal=# CREATE OR REPLACE FUNCTION car_portal_app.car_model3(model_name TEXT) RETURNS TABLE (car_model_id INT , make T
EXT) AS $$
car_portal$# BEGIN
car_portal$#   RETURN QUERY SELECT a.car_model_id, a.make FROM car_portal_app.car_model a WHERE model = model_name;
car_portal$# END;
car_portal$# $$ LANGUAGE plpgsql;
CREATE FUNCTION
car_portal=# SELECT * FROM car_portal_app.car_model3('A1');
 car_model_id | make
--------------+------
            1 | Audi
(1 row)
```

- The preceding function can also be written using the `OUT` variables, as follows:

```
car_portal=# CREATE OR REPLACE FUNCTION car_portal_app.car_model4(model_name TEXT, OUT car_model_id INT, OUT make TEXT ) RE
TURNS SETOF RECORD AS $$
car_portal$# BEGIN
car_portal$#   RETURN QUERY SELECT a.car_model_id, a.make FROM car_portal_app.car_model a WHERE model = model_name;
car_portal$# END;
car_portal$# $$ LANGUAGE plpgsql;
CREATE FUNCTION
car_portal=# SELECT * FROM car_portal_app.car_model4('A1'::text);
 car_model_id | make
--------------+------
            1 | Audi
(1 row)
```