PostgreSQL PL/pgSQL

- Based on ADA
- Assignments: direct (:=) and retrieval (INTO)
- Conditional Statements:

Oracle PL/SQL: Same but with { }

```
IF <condition>
THEN
      <statement;>
ELSE
      <statement;>
END IF;
```

```
IF <condition>
THEN
      <statement;>
ELSIF <condtion>
THEN
      <statement;>
ELSE
      <statement;>
END IF;
```


PostgreSQL PL/pgSQL

- Iterative Statements

Oracle PL/SQL: Same but with { }

  - Simple Loop

```
LOOP
   <statement;>
   EXIT; [or EXIT WHEN condition;]
END LOOP;
```

  - While Loop

```
WHILE <condition> LOOP <statement;> END LOOP;
```

  - For Loop

```
FOR counter IN [REVERSE] val1..val2 LOOP
    <statement;
END LOOP;
```


ANSI Exceptions

- Signaling of exception conditions, and declaring handlers for exceptions

```
declare out_of_stock condition
declare exit handler for out_of_stock
begin
…
  .. signal out_of_stock
end;
```

- The handler here is **exit** -- causes enclosing begin..end to be exited
- Other actions possible on exception

Exception handling in PL/pgSQL

- EXCEPTION clause at the end of a block:

```
BEGIN
  statements;
    …
EXCEPTION
   WHEN condition [ OR condition … ] THEN
        handler_statements;
   [ WHEN condition [ OR condition … ] THEN
        handler_statements;]
        …
   [ WHEN OTHERS THEN
        handler_statements;]
 END;
```

- PostgreSQL Error Codes: https://www.postgresql.org/docs/12/errcodes-appendix.html

PL/pgSQL: Block Structure

Student(SID,Name,Major,QPA)

```
DO $$                          -- anonymous function
Declare                        -- optional
  x integer := 0;
  y student.sid%type;
Begin                          -- mandatory
  select count(*) into x
  from Student
  where Major = 'CS';          -- RAISE: report messages and raise errors
  if x < 1 then RAISE 'bad_data' USING errcode='MYERR';   -- User defined error code
  else RAISE notice 'Number of CS Majors =%',x;           -- (5 characters)
  end if;
Exception                      -- optional
   when sqlstate 'MYERR' then
      RAISE notice 'troubles';
End $$;
```


PL/SQL: Var & Const

- **DECLARE**: introduces variables, constrants & records
- **Variables & Constants**
  <variable_name> datatype [NOT NULL := value];
  <constant_name> CONSTANT datatype := VALUE;
- Declaration of variables/constants based on a column from database table
  <variable_name> table_name.column_name%type;
  - E.g., y student.SID%type;

PL/SQL: Records

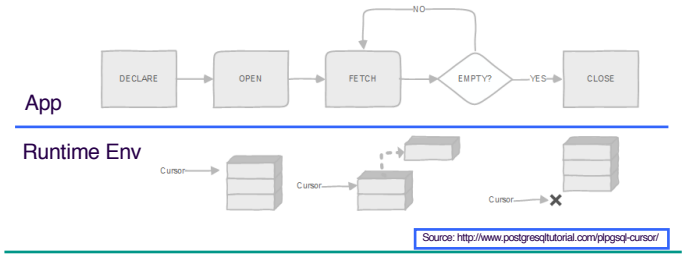
- **Record type**
  TYPE <record_type_name> IS RECORD
  (<1st_col_name> datatype,
  <2nd_col_name> datatype, ...);
  - Declare fields based on a column from database table
    col_name table_name.column_name%type;
- **Record variable declaration**
  - User-defined: record_name record_type_name;
  - DB-based: record_name table_name%ROWTYPE;
  - E.g., student_rec Student%rowtype;

PL/pgSQL: "Record"

- *rec_name* **RECORD;**
  - Has no predefined structure
  - Substructure is set when it is assigned a value

Cursors: Multiple Tuple Retrieval

- If more than one tuples can be selected, then tuples must be processed one at a time by means of a cursor
  - This is similar to the record-at-a-time processing
- A cursor is a "pointer" to a tuple in a result of a query
  - Current tuple w.r.t. a cursor is the tuple pointed by the cursor



Source: http://www.postgresqltutorial.com/plpgsql-cursor/

PL/pgSQL Cursors

- <cursor_name> *CURSOR* {IS | FOR} <query>
  - It declares a cursor by defining a query to be associated with a cursor with it
  - E.g.,
```
curs1 CURSOR FOR SELECT * FROM table1;
curs2 CURSOR (key integer) IS
      SELECT * FROM table1 WHERE att1 = key;
```
- *OPEN* <cursor_name> brings the query result from the DB and positions the cursor before the first tuple
  - Before a cursor can be used, it must be opened
    - `OPEN curs1;`
    - `OPEN curs2(42);`
    - `OPEN curs2(key:=42);`
- *CLOSE* <cursor_name> closes the named cursor and deletes the associated result table

PL/pgSQL Cursors

- *Fetch* copies into variables the current tuple and advances the cursor

  `FETCH [direction { FROM | IN }] cursor INTO target;`
  - *target* should be a RECORD (ROWTYPE) or list of variables
  - *direction* can take on many forms, e.g.:
    - `FETCH curs1 INTO student_rec;`
    - `FETCH curs2 INTO SID, Name, Major, QPA;`
    - `FETCH LAST FROM curs3 INTO x, y;`
    - `FETCH RELATIVE -2 FROM curs4 INTO recvar;`
  - Special variable **FOUND** will be set to true if a row is returned from the fetch
- *Move* repositions a cursor without retrieving any data

  `MOVE [ direction { FROM | IN } ] cursor;`

Cursor example in Postgress

```
CREATE FUNCTION qpa_avg() RETURNS DECIMAL AS $$
DECLARE
qpa_sum DECIMAL := 0; st_count INTEGER := 0;
st_cursor CURSOR FOR SELECT SID, Name, Major, QPA
                     FROM Student;
student_rec Student%ROWTYPE;
BEGIN
    OPEN st_cursor;
    LOOP
        FETCH st_cursor INTO student_rec;
        IF NOT FOUND THEN
            EXIT;
        END IF;
        qpa_sum := qpa_sum + student_rec.QPA;
        st_count := st_count + 1;
    END LOOP;
    CLOSE st_cursor; RETURN qpa_sum / st_count;
END; $$ LANGUAGE plpgsql;
```

select qpa_avg();

Student(SID,Name,Major,QPA)

Implicit Cursor in Postgress

```
CREATE OR REPLACE PROCEDURE proc_retrieve_reservations()
    LANGUAGE plpgsql
AS
$$
DECLARE
    reservation_record RECORD;
BEGIN
  -- Loop across all reservation numbers & prints them out
  FOR reservation_record IN SELECT * FROM Reservation
  LOOP
   raise notice '%',reservation_record.Reservation_Number;
  END LOOP;
END
$$;
```


External Language Functions/Procedures

- Declaring external language procedures and functions
  - In C/C++


```
create procedure author_count_proc (in title varchar(20),
                                    out count integer)
language C
external name ' /usr/db/bin/author_count_proc'
```

  - In Java

```
create function author_count ( title varchar(20) )
  returns integer
language Java
external name '/usr/db/bin/author_count.jar'
```

Example in PostgreSQL for C/C++

- Supposing that the increment function was implemented in file funcs.c and compiled into a shared object
  - cc -fPIC -c funcs.c
  - cc -shared -o funcs.so funcs.o
- The increment function is introduced to PostgreSQL:

```
CREATE FUNCTION add_one(integer) RETURNS integer
    AS 'DIRECTORY/funcs', 'add_one'
    LANGUAGE C STRICT;
```

- Then it can be called as usual:

```
SELECT name, add_one(age) AS new_age
     FROM emp
     WHERE name = 'Bill' OR name = 'Sam';
```


External Routines: Performance Vs. Security

- Benefits of external language functions/procedures:
  - more efficient for many operations, and
  - more expressive power
- Drawbacks
  - Code to implement function may need to be executed in the database system's address space
    - risk of accidental corruption of database structures
    - security risk, allowing users access to unauthorized data

  - Use *sandbox* techniques
    - that is use a safe language like Java
- Direct execution in the database system's space is used when efficiency is more important than security

Disadvantages of stored procedures

- Slowness in software development because stored procedure programming requires specialized skills that many developers do not possess.
- Difficult to manage versions and hard to debug.
- May not be portable to other database management systems e.g., MySQL or Microsoft SQL Server.

Database Programming Approaches

- Embedded commands:
  - Database commands are **embedded** in a general-purpose programming language
- Library of database functions:
  - Available to the host language for database calls; known as an **API** (Application Program Interface)
  - *e.g., JDBC, ODBC, PHP, Python*
- A brand new, full-fledged language
  - e.g., Oracle PL/SQL, Postgres PL/pgSQL
  - **P**rocedural **L**anguage extensions to SQL

Embedded SQL (ESQL)

- SQL statements are embedded by enclosing them:
  - between "&SQL(" and ")";
  - between "EXEC SQL" and "END-EXEC"; or
  - between "EXEC SQL" and ";"
- E.g., **EXEC SQL** DELETE FROM STUDENT
  WHERE Name LIKE 'John %';
- Two types of statement-level embedding:
  - *Static SQL*: complete SQL statements
  - *Dynamic SQL*: statements are created during execution time

Structure of ESQL Programs

1. Define host data structures
2. Open database using EXEC SQL CONNECT dbname/username IDENTIFIED BY password
3. Start a transaction using EXEC SQL SET TRANSACTION command
4. Retrieve data using EXEC SQL SELECT and load into data structure that overlays row
5. Write data back to the database using EXEC SQL UPDATE, or INSERT, or DELETE
6. Terminate the transaction using either EXEC SQL COMMIT or EXEC SQL ROLLBACK
7. Close database using EXEC SQL DISCONNECT

Host Data Structure

- Program variables used within an SQL command are declared within a DECLARE SECTION
- E.g.,

```
EXEC SQL BEGIN DECLARE SECTION
char student_name[20];
EXEC SQL END DECLARE SECTION
```

- Host structures/records (e.g., C struct) must match tuple formats *exactly*
  - field order is important
  - strings in C/C++ are terminated with NULL character
  - e.g., if Student.Major char(4), then char major[5] in C/C++

Scope Rules

- The arguments used in an SQL statement could be *constants* or program *variables*
- Program variables within an SQL command are prefixed with "**:**"
- E.g.,

```
EXEC SQL BEGIN DECLARE SECTION
char student_name[20];
EXEC SQL END DECLARE SECTION

cout << "Please Enter Student Name to be deleted" << endl;
cin >> (char *) student_name;
EXEC SQL DELETE FROM STUDENT
        WHERE Name = : student_name;
```


Status of an SQL Cmd Execution

- SQLCODE in SQL1 is an integer variable containing the Status Code returned by SQL/DBS
  - Zero (0) if SQL command is successful
  - Nonzero positive if SQL generates a warning
    - 100: data not found
  - Negative if SQL command fails (error)
    - -913: resource deadlock
- SQLSTATE in SQL2 is a 5-character string
    - 02000: data not found