# Lecture 12: Semantic Integrity Constraints

## CS 1555: Database Management Systems

**Constantinos Costa**

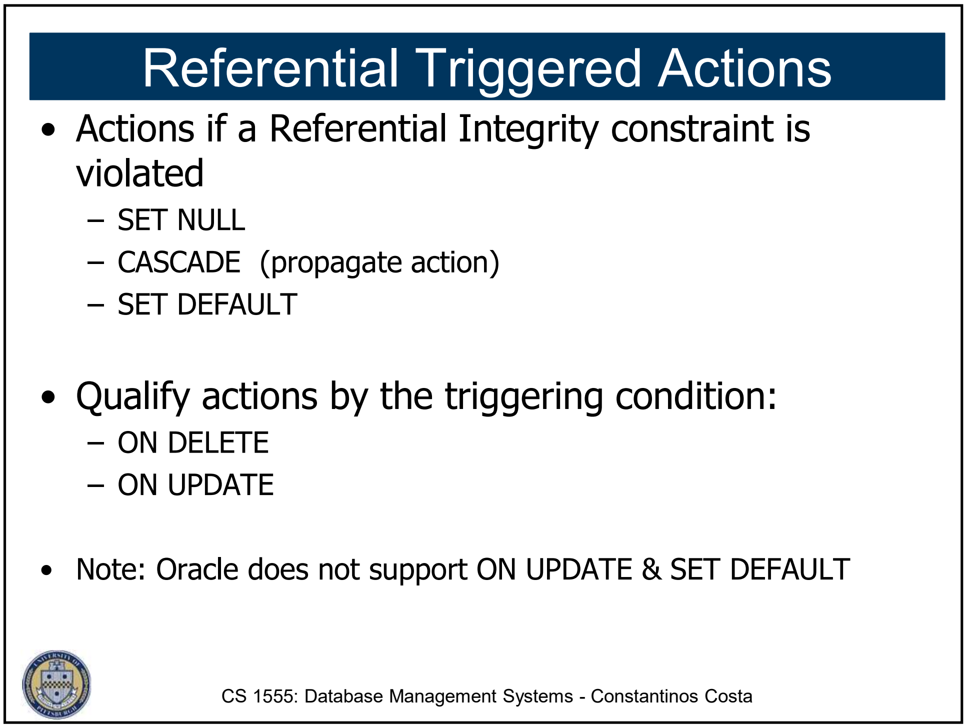
http://db.cs.pitt.edu/courses/cs1555/current.term/

March 07, 2019, 16:00-17:15
University of Pittsburgh, Pittsburgh, PA

Lectures based: P. Chrysanthis & N. Farnan Lectures

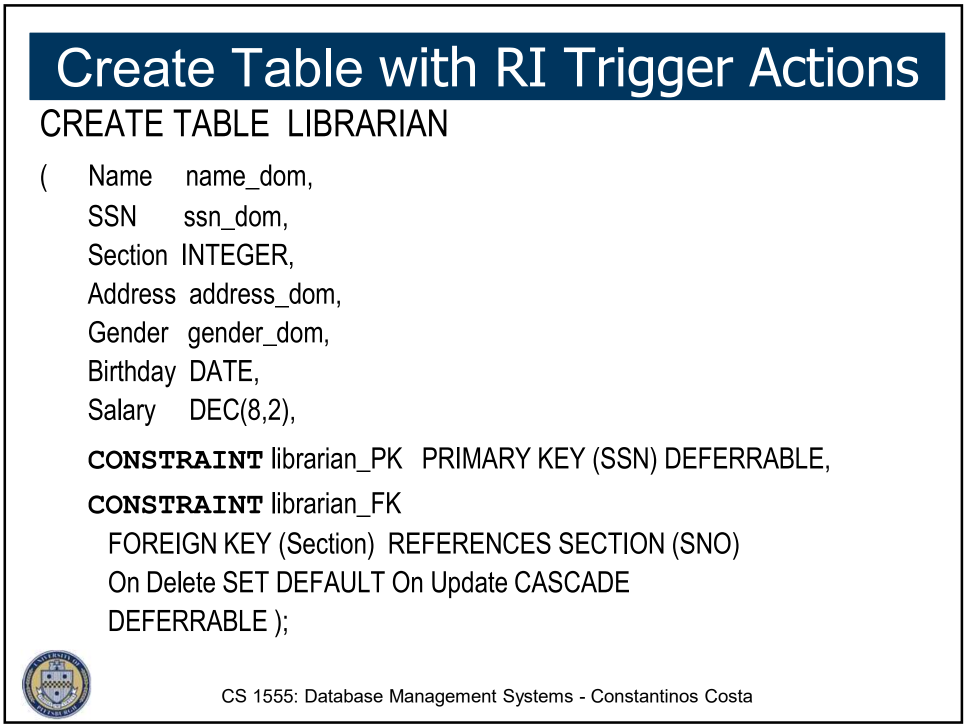
# Structural Constraints

- Constraints (on Attributes):
  - NOT NULL
  - DEFAULT value
    - without the DEFAULT-clause, the default value is NULL
  - PRIMARY KEY ( attribute-list )
  - UNIQUE ( attribute list )
    - allows the specification of alternative key
  - FOREIGN KEY (key) REFERENCES table (key)



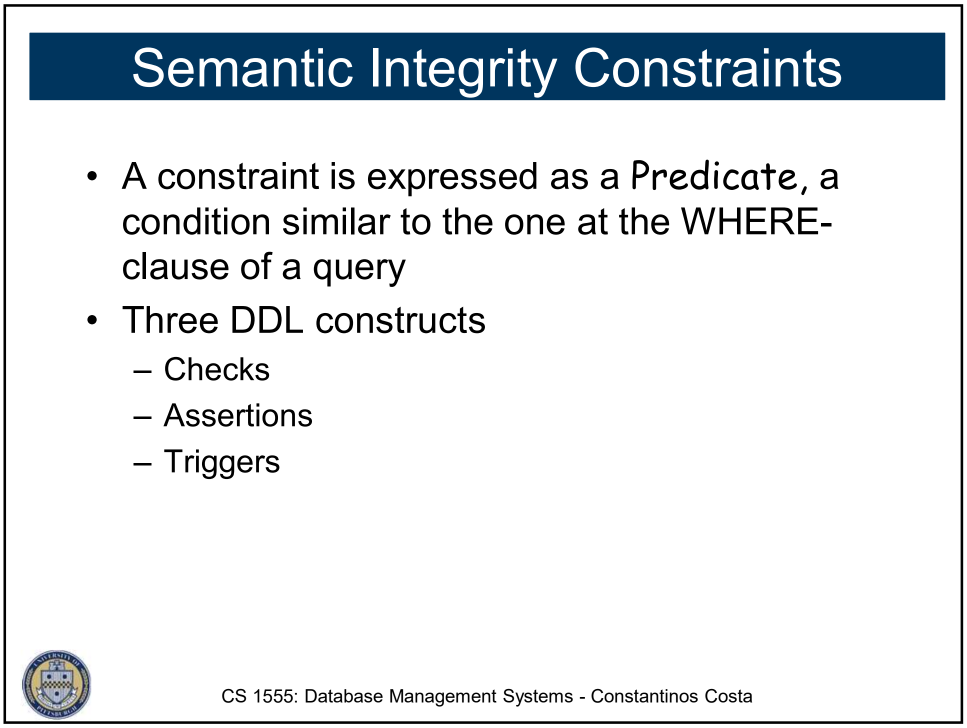
# Referential Triggered Actions

- Actions if a Referential Integrity constraint is violated
  - SET NULL
  - CASCADE (propagate action)
  - SET DEFAULT
- Qualify actions by the triggering condition:
  - ON DELETE
  - ON UPDATE
- Note: Oracle does not support ON UPDATE & SET DEFAULT



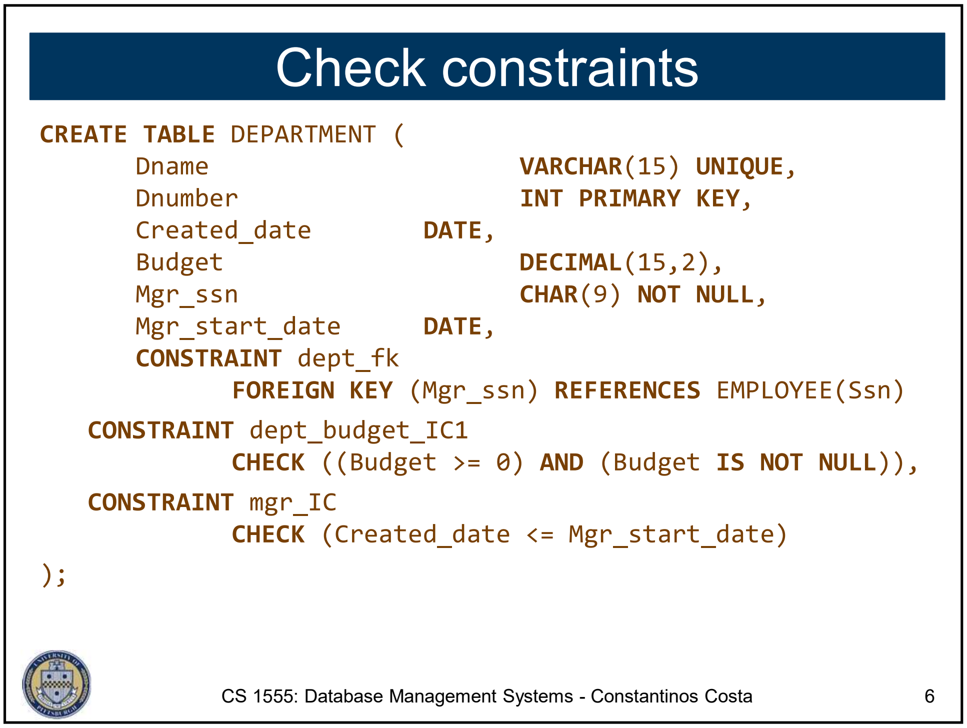
# Create Table with RI Trigger Actions

```
CREATE TABLE  LIBRARIAN
(     Name     name_dom,
      SSN      ssn_dom,
      Section  INTEGER,
      Address  address_dom,
      Gender   gender_dom,
      Birthday DATE,
      Salary   DEC(8,2),
      CONSTRAINT librarian_PK   PRIMARY KEY (SSN) DEFERRABLE,
      CONSTRAINT librarian_FK
        FOREIGN KEY (Section)  REFERENCES SECTION (SNO)
        On Delete SET DEFAULT On Update CASCADE
        DEFERRABLE );
```



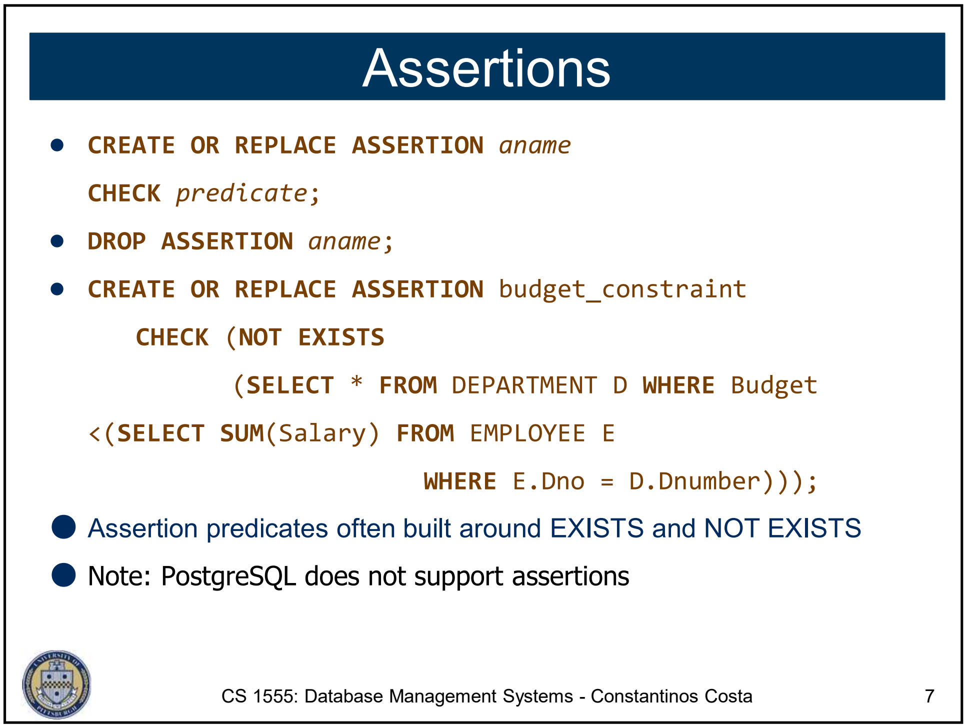
## Semantic Integrity Constraints

- A constraint is expressed as a *Predicate*, a condition similar to the one at the WHERE-clause of a query
- Three DDL constructs
  - Checks
  - Assertions
  - Triggers



## Check constraints

```
CREATE TABLE DEPARTMENT (
      Dname                         VARCHAR(15) UNIQUE,
      Dnumber                       INT PRIMARY KEY,
      Created_date            DATE,
      Budget                        DECIMAL(15,2),
      Mgr_ssn                       CHAR(9) NOT NULL,
      Mgr_start_date          DATE,
      CONSTRAINT dept_fk
             FOREIGN KEY (Mgr_ssn) REFERENCES EMPLOYEE(Ssn)
   CONSTRAINT dept_budget_IC1
             CHECK ((Budget >= 0) AND (Budget IS NOT NULL)),
   CONSTRAINT mgr_IC
             CHECK (Created_date <= Mgr_start_date)
);
```

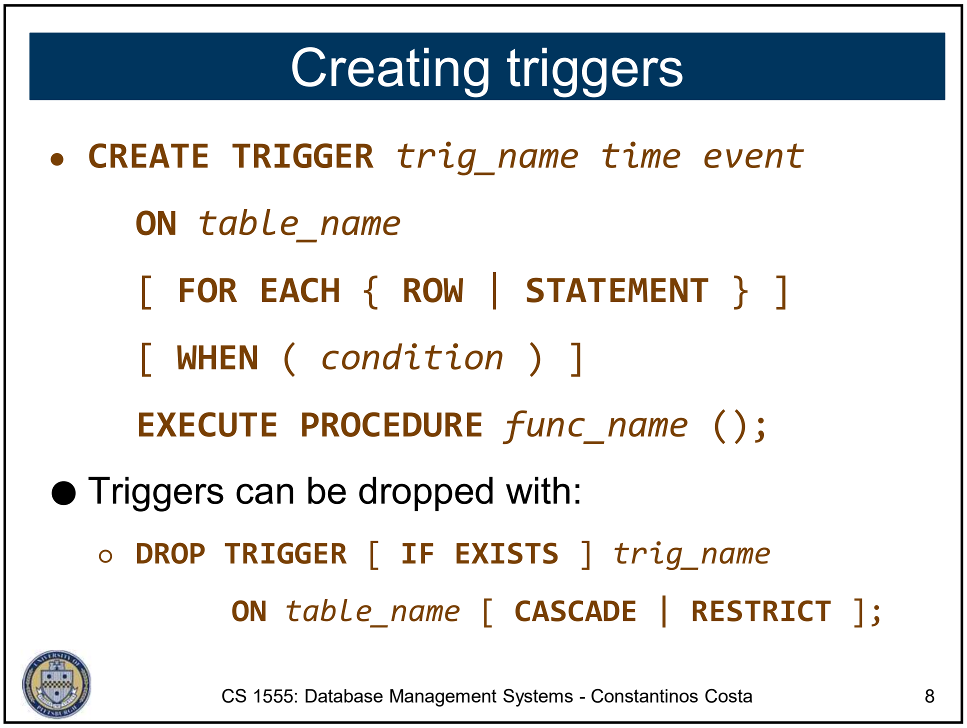


# Assertions

-
  ```
  CREATE OR REPLACE ASSERTION aname
  CHECK predicate;
  ```

-
  ```
  DROP ASSERTION aname;
  ```

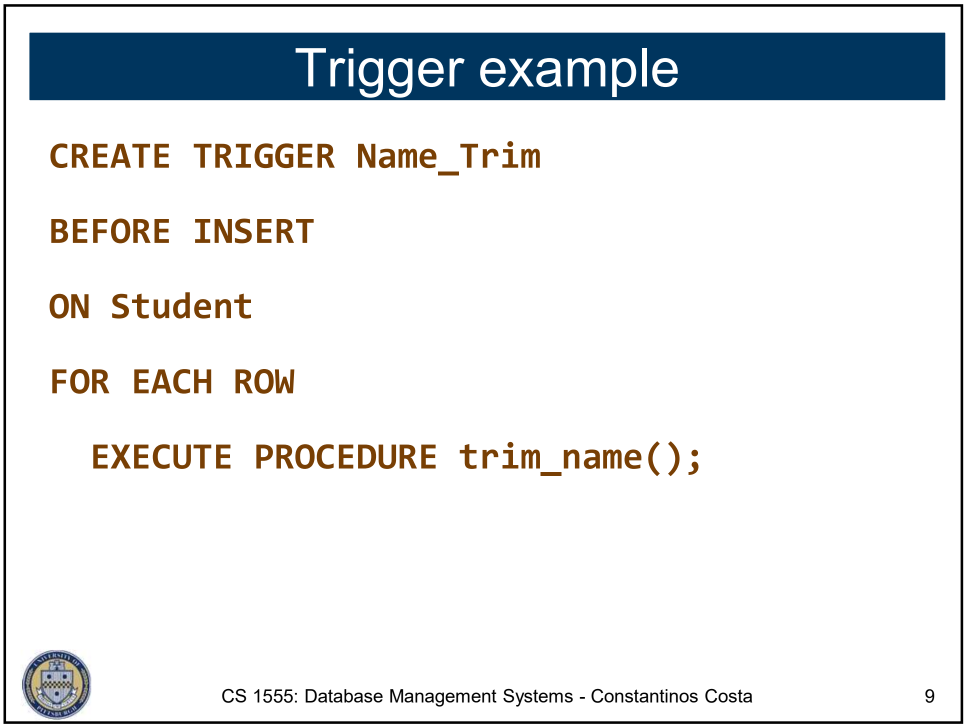
-
  ```
  CREATE OR REPLACE ASSERTION budget_constraint
     CHECK (NOT EXISTS
           (SELECT * FROM DEPARTMENT D WHERE Budget
  <(SELECT SUM(Salary) FROM EMPLOYEE E
                      WHERE E.Dno = D.Dnumber)));
  ```
- Assertion predicates often built around EXISTS and NOT EXISTS
- Note: PostgreSQL does not support assertions



# Creating triggers

-
  ```
  CREATE TRIGGER trig_name time event
     ON table_name
     [ FOR EACH { ROW | STATEMENT } ]
     [ WHEN ( condition ) ]
     EXECUTE PROCEDURE func_name ();
  ```
- Triggers can be dropped with:

  -
    ```
    DROP TRIGGER [ IF EXISTS ] trig_name
          ON table_name [ CASCADE | RESTRICT ];
    ```



## Trigger example

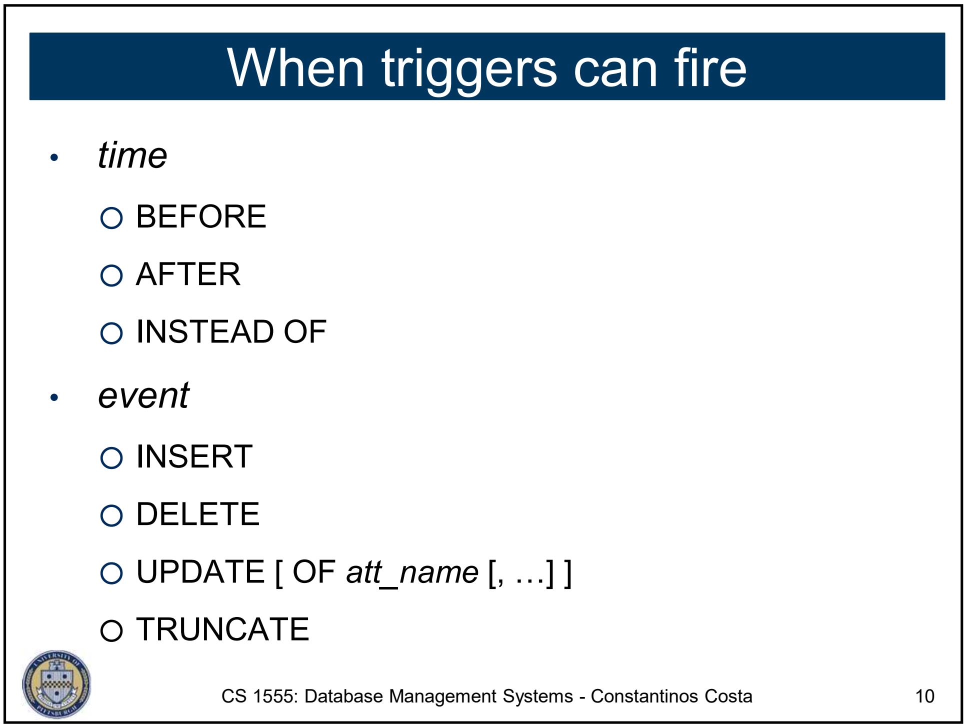
```
CREATE TRIGGER Name_Trim

BEFORE INSERT

ON Student

FOR EACH ROW

  EXECUTE PROCEDURE trim_name();
```



## When triggers can fire

- *time*
  - BEFORE
  - AFTER
  - INSTEAD OF
- *event*
  - INSERT
  - DELETE
  - UPDATE [ OF *att_name* [, …] ]
  - TRUNCATE

# Compatibility

| WHEN | EVENT | ROW | STATEMENT |
|---|---|---|---|
| BEFORE | INSERT, UPDATE, DELETE | Tables | Tables and views |
| | TRUNCATE | — | Tables |
| AFTER | INSERT, UPDATE, DELETE | Tables | Tables and views |
| | TRUNCATE | — | Tables |
| INSTEAD OF | INSERT, UPDATE, DELETE | Views | — |
| | TRUNCATE | — | — |

# A few more trigger points

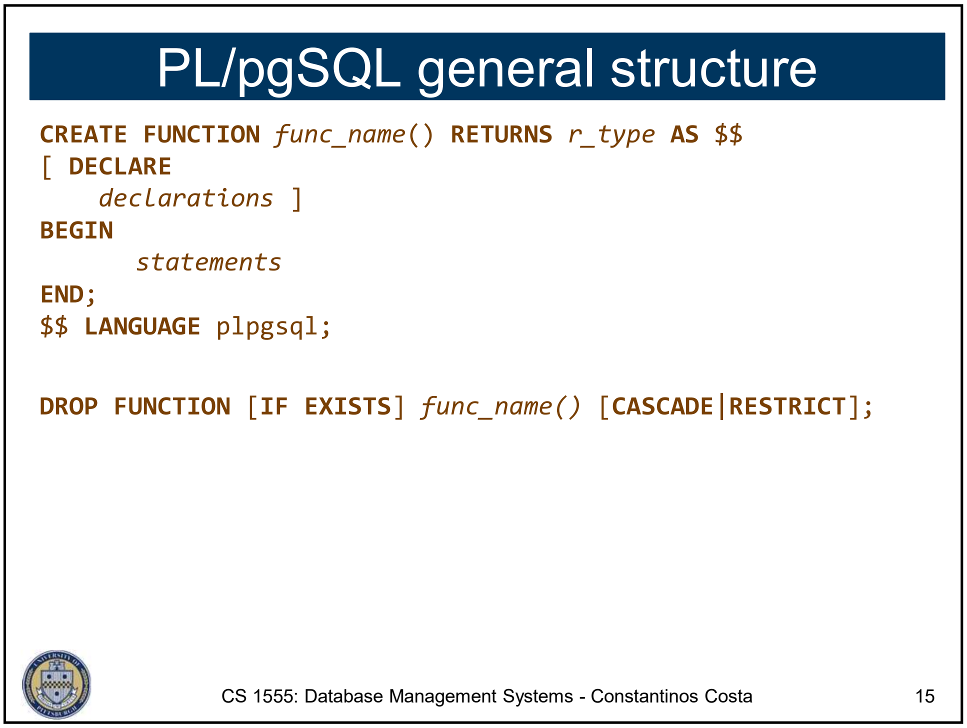
- FOR EACH ROW classifies the trigger as a *row trigger*
  - In contrast to a *statement trigger*
  - E.g., if an SQL UPDATE updates 10 rows, a row trigger fires 10 times, a statement trigger fires once
- NEW and OLD allow reference to be made to new and old values of tuples affected by the trigger (and satisfy the WHEN clause) in a row trigger
- WHEN specifies a condition that must be met for the trigger to fire

# Trigger maintenance

-
  ```
  ALTER TABLE tname
      { ENABLE | DISABLE } [ trigger_name | ALL | USER ];
  ```

# Trigger procedures

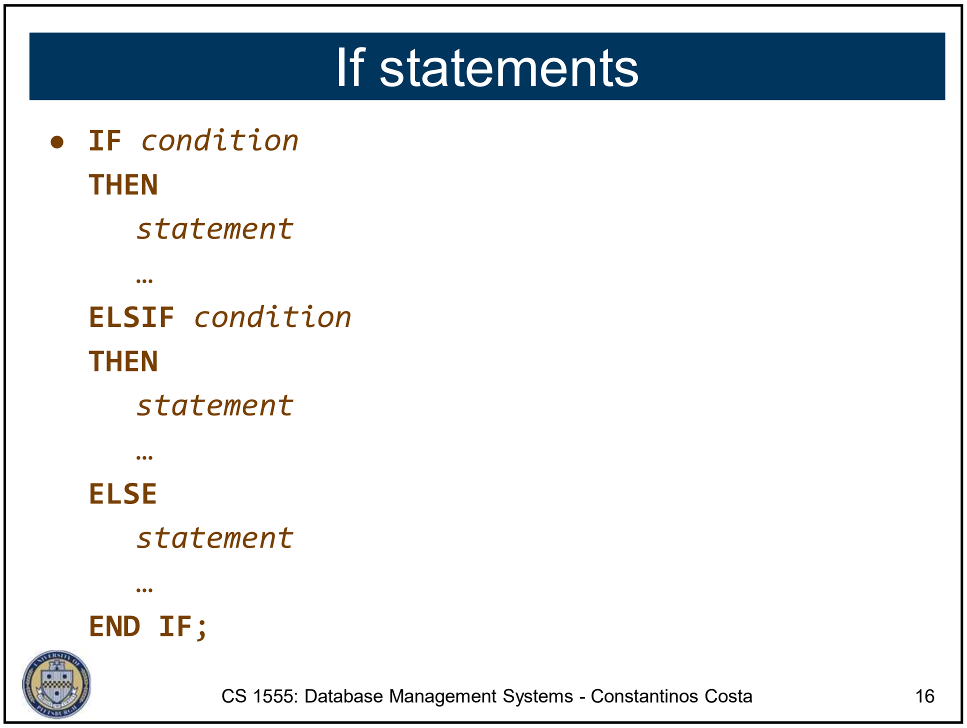
- How can we define procedures in SQL??
  - We can't, we need a procedural language to do this
    - PL/pgSQL
      - The procedural language for PostgreSQL



# PL/pgSQL general structure

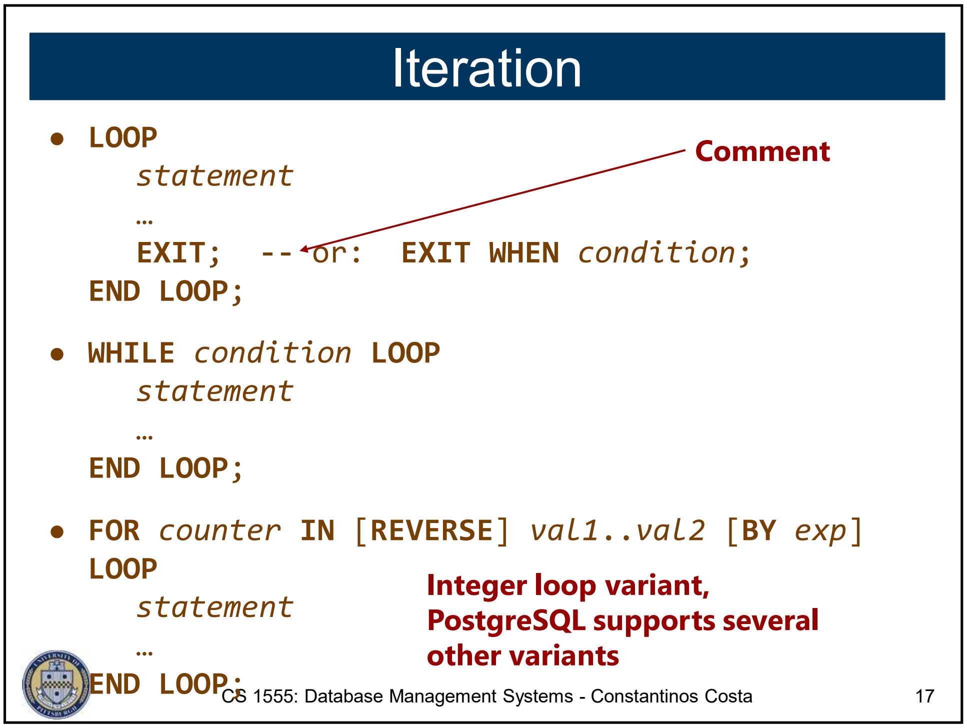
```
CREATE FUNCTION func_name() RETURNS r_type AS $$
[ DECLARE
    declarations ]
BEGIN
      statements
END;
$$ LANGUAGE plpgsql;

DROP FUNCTION [IF EXISTS] func_name() [CASCADE|RESTRICT];
```



# If statements

-
  ```
  IF condition
  THEN
     statement
     …
  ELSIF condition
  THEN
     statement
     …
  ELSE
     statement
     …
  END IF;
  ```



## Iteration

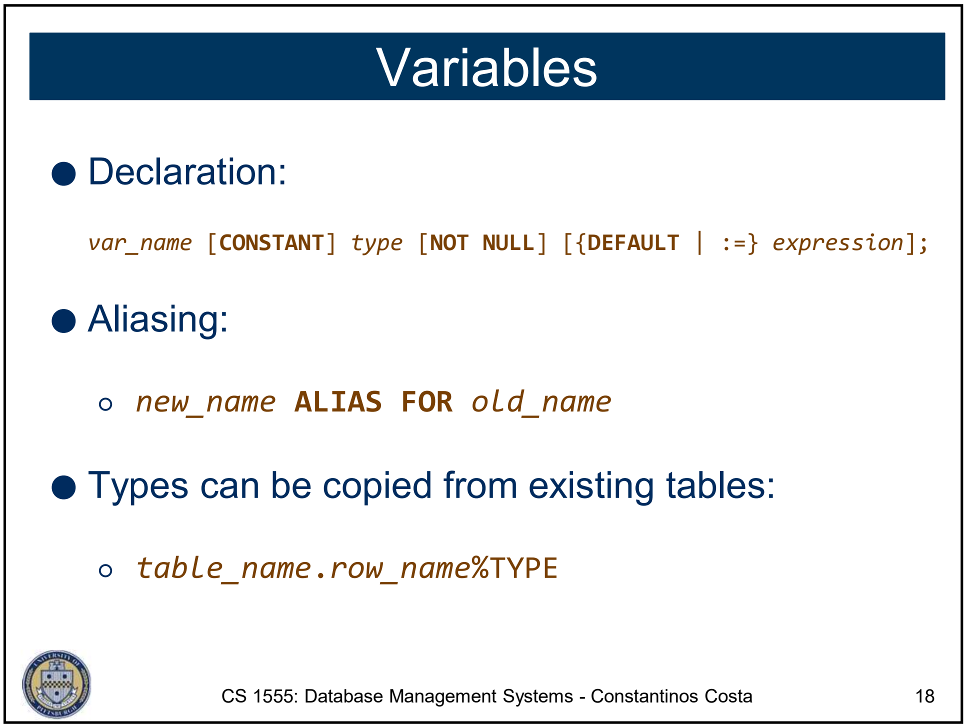
```
LOOP
   statement
   …
   EXIT;  -- or:  EXIT WHEN condition;
END LOOP;
```

Comment

```
WHILE condition LOOP
   statement
   …
END LOOP;
```

```
FOR counter IN [REVERSE] val1..val2 [BY exp]
LOOP
   statement
   …
END LOOP;
```

Integer loop variant, PostgreSQL supports several other variants



## Variables

- Declaration:

```
var_name [CONSTANT] type [NOT NULL] [{DEFAULT | :=} expression];
```

- Aliasing:
  - `new_name ALIAS FOR old_name`
- Types can be copied from existing tables:
  - `table_name.row_name%TYPE`

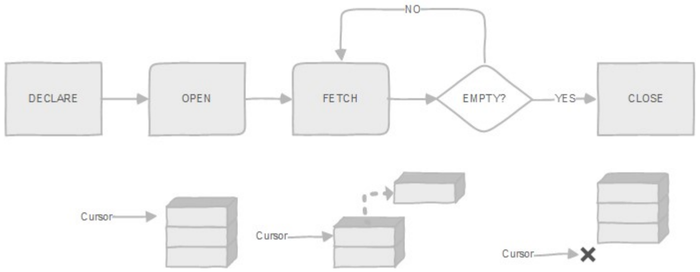
# Processing relations

- How can we represent relations in a procedural context?

# Records: storing table rows

- `rec_name RECORD;`
  - Has no predefined structure
  - Substructure is set when it is assigned a value
- `rec_name table_name%ROWTYPE;`
  - Structured to match the schema of *table_name*

# Cursors

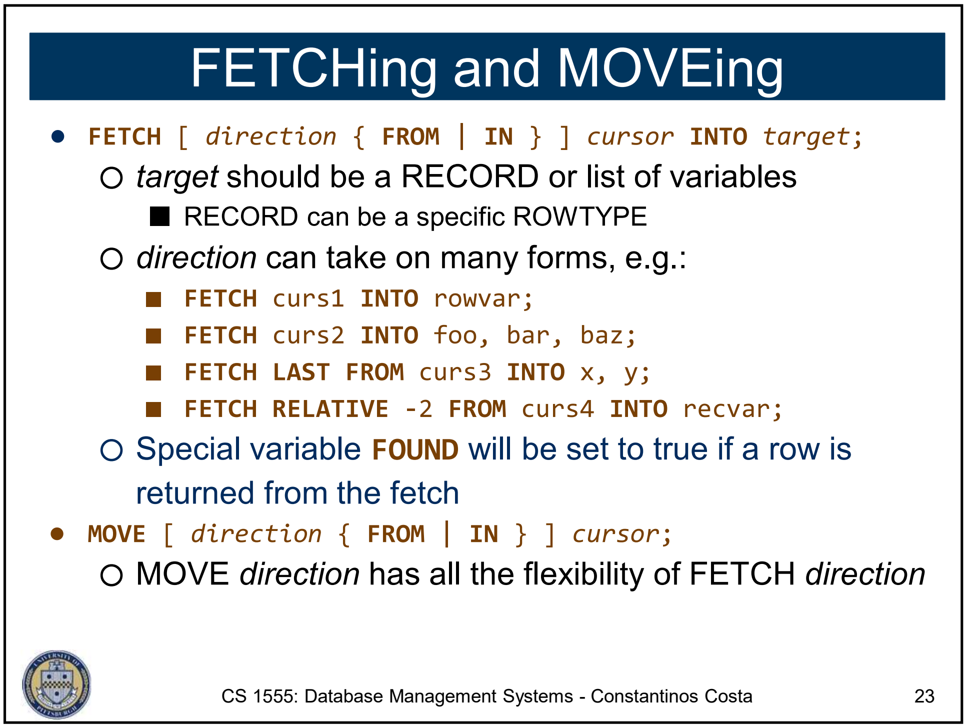


**Source: http://www.postgresqltutorial.com/plpgsql-cursor/**

# Cursors

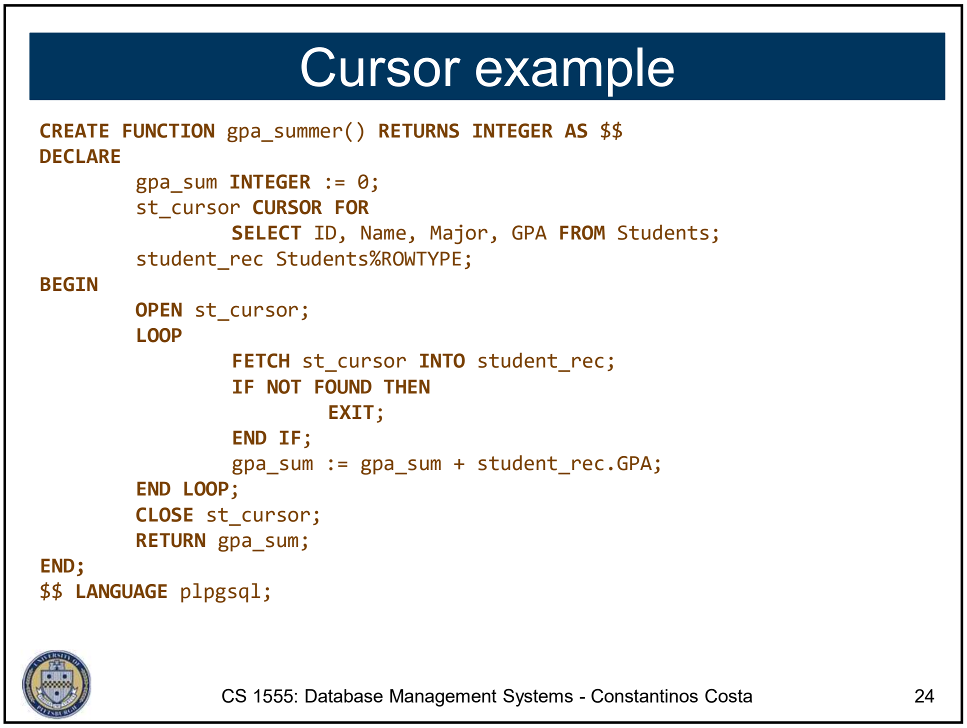
- `cur_name [ [ NO ] SCROLL ] CURSOR [ ( args ) ] FOR query;`
  - E.g.:
    - `curs1 CURSOR FOR SELECT * FROM table1;`

    -
      ```
      curs2 CURSOR (key integer) FOR
          SELECT * FROM table1 WHERE att1 = key;
      ```
- Before a cursor can be used, it must be opened
  - `OPEN curs1;`
  - `OPEN curs2(42);`
  - `OPEN curs2(key := 42);`
- Should be closed when finished:
  - `CLOSE cur_name;`



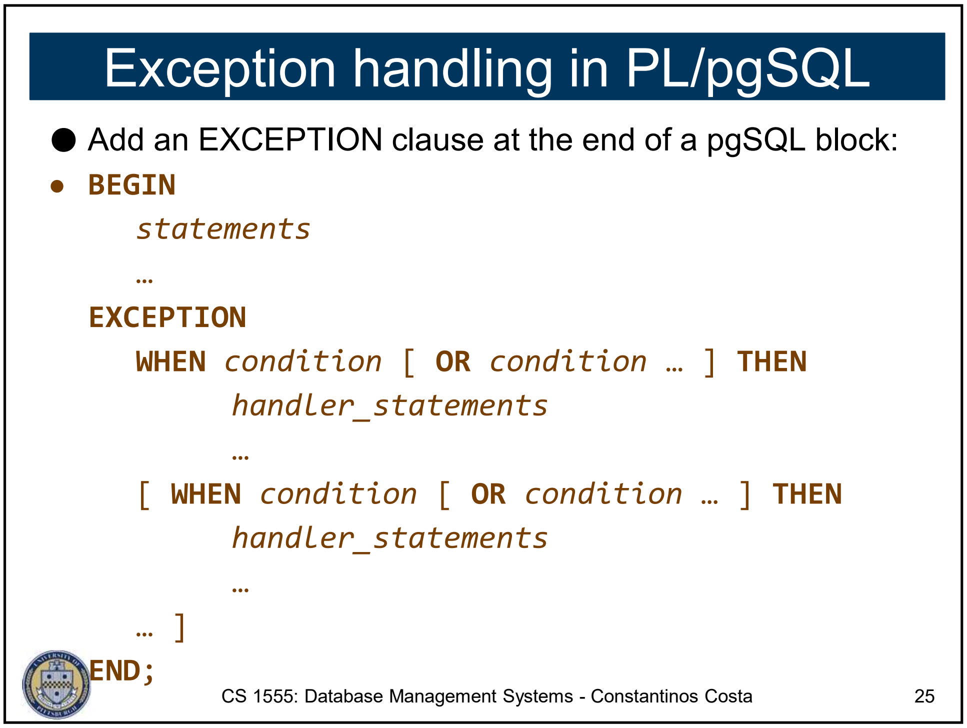
## FETCHing and MOVEing

- `FETCH [ direction { FROM | IN } ] cursor INTO target;`
  - ○ *target* should be a RECORD or list of variables
    - ■ RECORD can be a specific ROWTYPE
  - ○ *direction* can take on many forms, e.g.:
    - ■ `FETCH curs1 INTO rowvar;`
    - ■ `FETCH curs2 INTO foo, bar, baz;`
    - ■ `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 [ direction { FROM | IN } ] cursor;`
  - ○ MOVE *direction* has all the flexibility of FETCH *direction*



## Cursor example

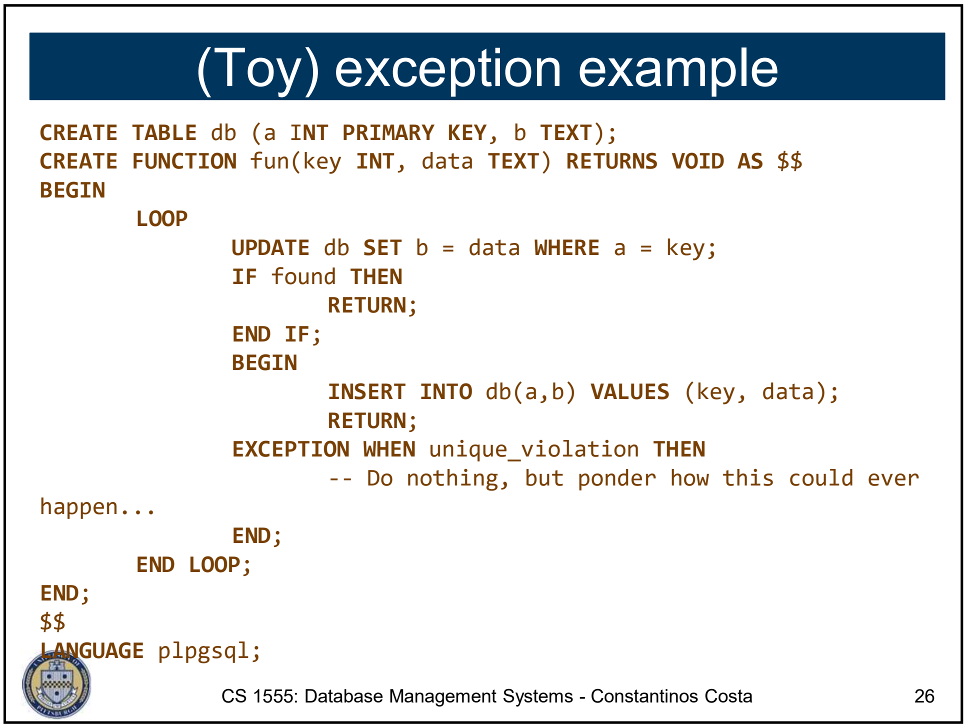
```
CREATE FUNCTION gpa_summer() RETURNS INTEGER AS $$
DECLARE
        gpa_sum INTEGER := 0;
        st_cursor CURSOR FOR
                SELECT ID, Name, Major, GPA FROM Students;
        student_rec Students%ROWTYPE;
BEGIN
        OPEN st_cursor;
        LOOP
                FETCH st_cursor INTO student_rec;
                IF NOT FOUND THEN
                        EXIT;
                END IF;
                gpa_sum := gpa_sum + student_rec.GPA;
        END LOOP;
        CLOSE st_cursor;
        RETURN gpa_sum;
END;
$$ LANGUAGE plpgsql;
```



# Exception handling in PL/pgSQL

- Add an EXCEPTION clause at the end of a pgSQL block:

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



# (Toy) exception example

```
CREATE TABLE db (a INT PRIMARY KEY, b TEXT);
CREATE FUNCTION fun(key INT, data TEXT) RETURNS VOID AS $$
BEGIN
        LOOP
                UPDATE db SET b = data WHERE a = key;
                IF found THEN
                        RETURN;
                END IF;
                BEGIN
                        INSERT INTO db(a,b) VALUES (key, data);
                        RETURN;
                EXCEPTION WHEN unique_violation THEN
                        -- Do nothing, but ponder how this could ever
happen...
                END;
        END LOOP;
END;
$$
LANGUAGE plpgsql;
```

# Getting back to triggers...

- PL/pgSQL code that will run when a trigger is fired must be stored in a *trigger procedure*
  - Has a return type of TRIGGER
  - Takes no arguments
  - Will have access to special variables
    - NEW
    - OLD
    - etc.
  - Can return replacement rows or NULL to indicate that an INSERT or UPDATE operation should not go through

# Assertions and Triggers

```
• CREATE OR REPLACE ASSERTION budget_constraint
     CHECK (NOT EXISTS
          (SELECT * FROM DEPARTMENT D WHERE
  Budget <(SELECT SUM(Salary) FROM EMPLOYEE E
                    WHERE E.Dno = D.Dnumber)));
```

- How can we create a trigger to accomplish the same functionality?

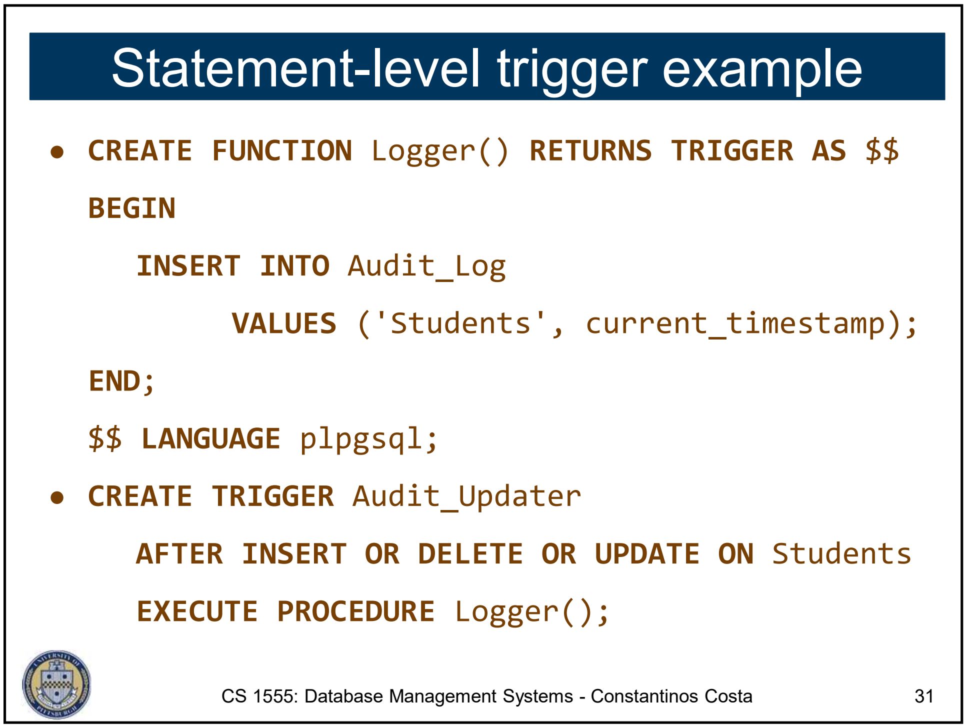
# Solution

-
  ```
  CREATE FUNCTION blocker() RETURNS TRIGGER AS $$
  BEGIN
      IF (EXISTS
                     (SELECT * FROM DEPARTMENT D WHERE Budget <
                             (SELECT SUM(Salary) FROM EMPLOYEE E
                                    WHERE E.Dno = D.Dnumber)))
      THEN
              RETURN NULL;
      END IF;
      RETURN NEW;
  END;
  $$ LANGUAGE plpgsql;
  ```

-
  ```
  CREATE TRIGGER test1
      BEFORE INSERT OR UPDATE OF Budget ON DEPARTMENT
      FOR EACH ROW EXECUTE PROCEDURE blocker();
  ```

# To ensure action is taken in either case

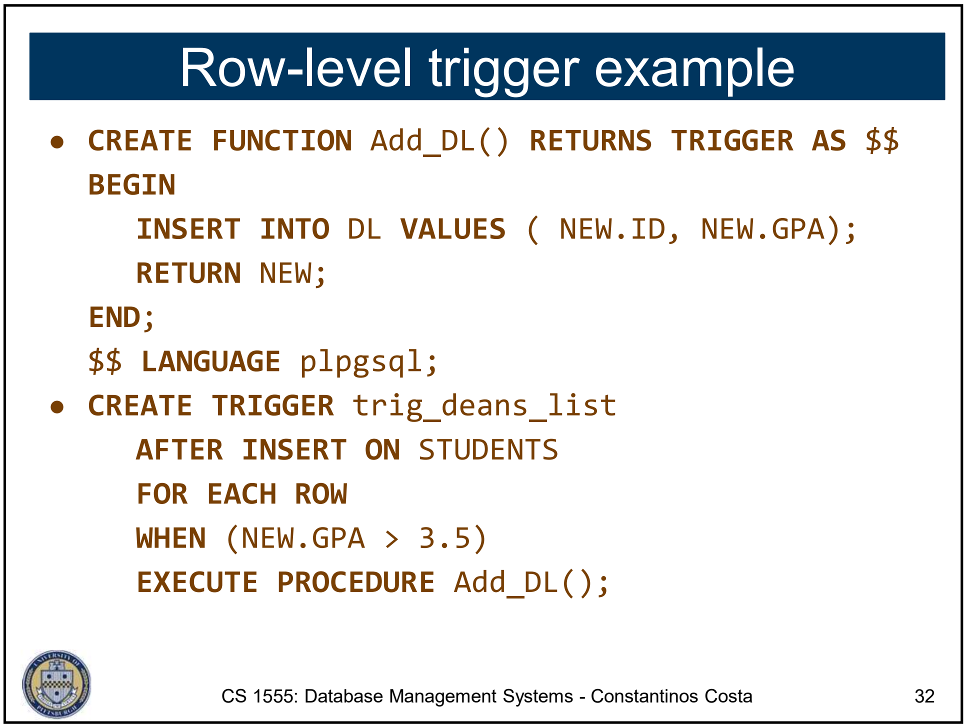
-
  ```
  CREATE TRIGGER test2
  BEFORE INSERT OR UPDATE OF Salary ON
  EMPLOYEE
  FOR EACH ROW EXECUTE PROCEDURE blocker();
  ```



# Statement-level trigger example

-
  ```
  CREATE FUNCTION Logger() RETURNS TRIGGER AS $$
  BEGIN
     INSERT INTO Audit_Log
           VALUES ('Students', current_timestamp);
  END;
  $$ LANGUAGE plpgsql;
  ```

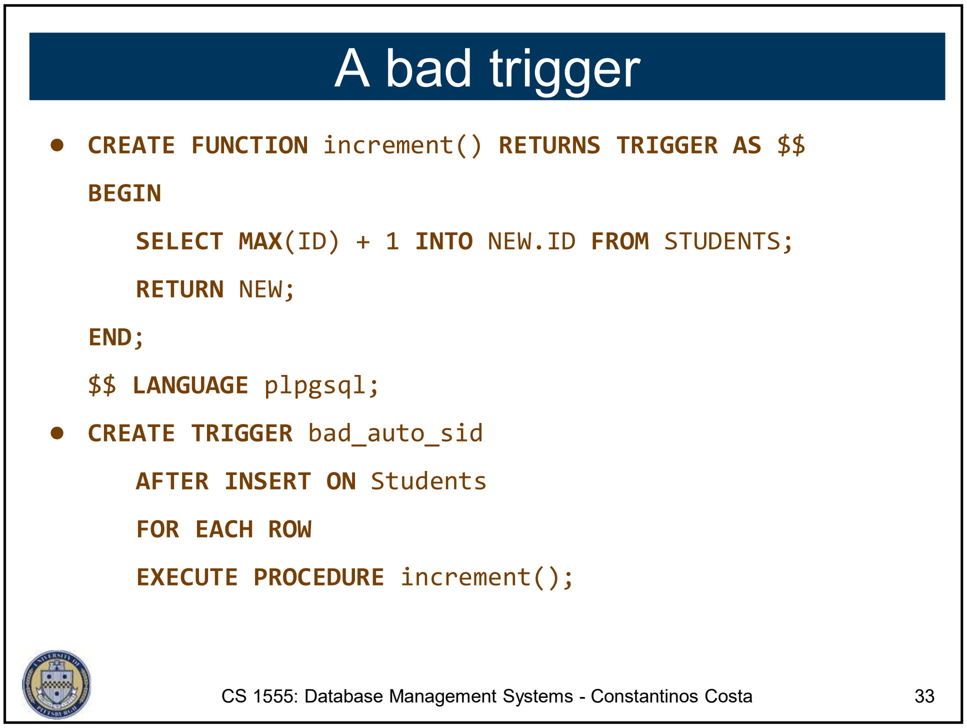
-
  ```
  CREATE TRIGGER Audit_Updater
     AFTER INSERT OR DELETE OR UPDATE ON Students
     EXECUTE PROCEDURE Logger();
  ```



# Row-level trigger example

-
  ```
  CREATE FUNCTION Add_DL() RETURNS TRIGGER AS $$
  BEGIN
     INSERT INTO DL VALUES ( NEW.ID, NEW.GPA);
     RETURN NEW;
  END;
  $$ LANGUAGE plpgsql;
  ```

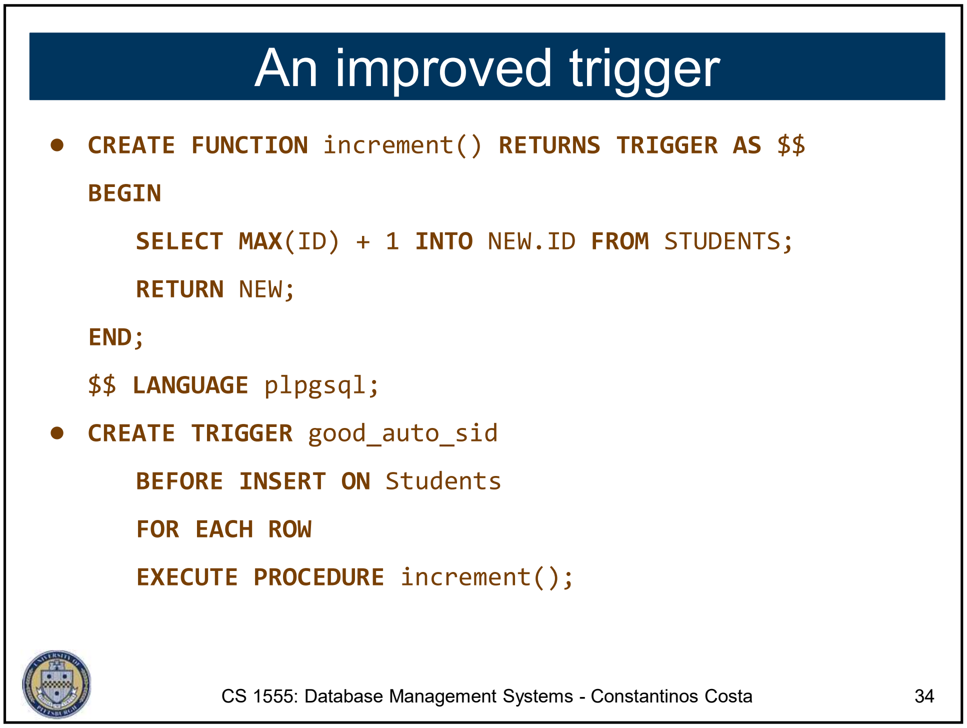
-
  ```
  CREATE TRIGGER trig_deans_list
     AFTER INSERT ON STUDENTS
     FOR EACH ROW
     WHEN (NEW.GPA > 3.5)
     EXECUTE PROCEDURE Add_DL();
  ```



# A bad trigger

-
  ```
  CREATE FUNCTION increment() RETURNS TRIGGER AS $$
  BEGIN
      SELECT MAX(ID) + 1 INTO NEW.ID FROM STUDENTS;
      RETURN NEW;
  END;
  $$ LANGUAGE plpgsql;
  ```

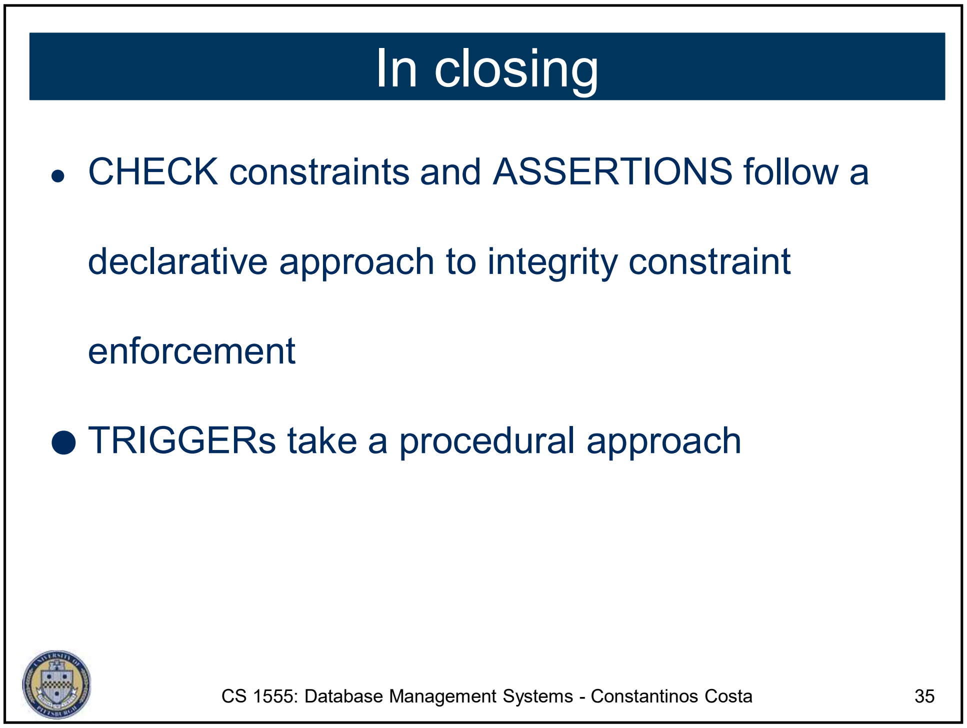
-
  ```
  CREATE TRIGGER bad_auto_sid
      AFTER INSERT ON Students
      FOR EACH ROW
      EXECUTE PROCEDURE increment();
  ```



# An improved trigger

-
  ```
  CREATE FUNCTION increment() RETURNS TRIGGER AS $$
  BEGIN
      SELECT MAX(ID) + 1 INTO NEW.ID FROM STUDENTS;
      RETURN NEW;
  END;
  $$ LANGUAGE plpgsql;
  ```

-
  ```
  CREATE TRIGGER good_auto_sid
      BEFORE INSERT ON Students
      FOR EACH ROW
      EXECUTE PROCEDURE increment();
  ```



# In closing

- CHECK constraints and ASSERTIONS follow a declarative approach to integrity constraint enforcement
- TRIGGERs take a procedural approach