

UNIT-3

PL/SQL

DATABASE APPLICATION DEVELOPMENT

Embedded SQL

❑ In real time applications we need access to a database from a general-purpose programming language.

For Example, to integrate a database application with a nice graphical user interface, or we may want to ask a query that cannot be expressed in SQL.

SQL provides a powerful declarative query language- **Embedded SQL**.

❑ The use of SQL commands within a host language program like c, java is called Embedded SQL.

❑ The SQL standard defines embedding of SQL as embedded SQL and the language in which SQL queries are embedded is referred as **host language**.

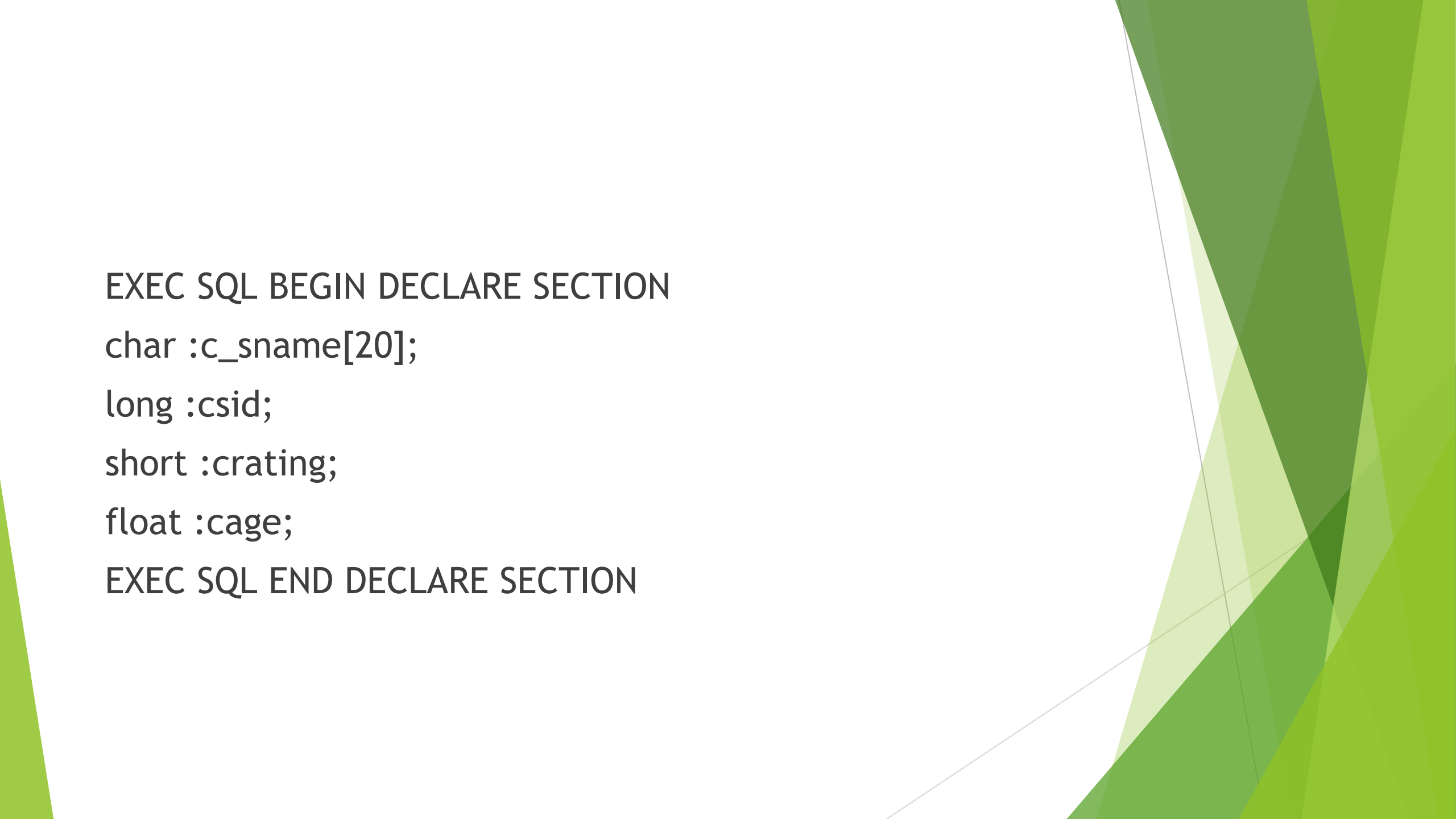
Complications

- ▶ First, the **data types recognized by SQL** may not be recognized by the host language and vice versa.
- ▶ This mismatch is typically addressed by casting data values appropriately before passing them to or from SQL commands.
- ▶ The second complication has to do with SQL being **set-oriented**, and is addressed using cursors.

Declaring Variables and Exceptions

- ▶ SQL statements can refer to variables defined in the host program.
- ▶ Host language variables must be prefixed by a colon (:) in SQL statements and be declared between the commands

EXEC SQL BEGIN DECLARE SECTION and EXEC SQL
END DECLARE SECTION.


The background features abstract, overlapping green geometric shapes, primarily triangles and polygons, in various shades of green, creating a modern, layered effect on the right side of the slide.

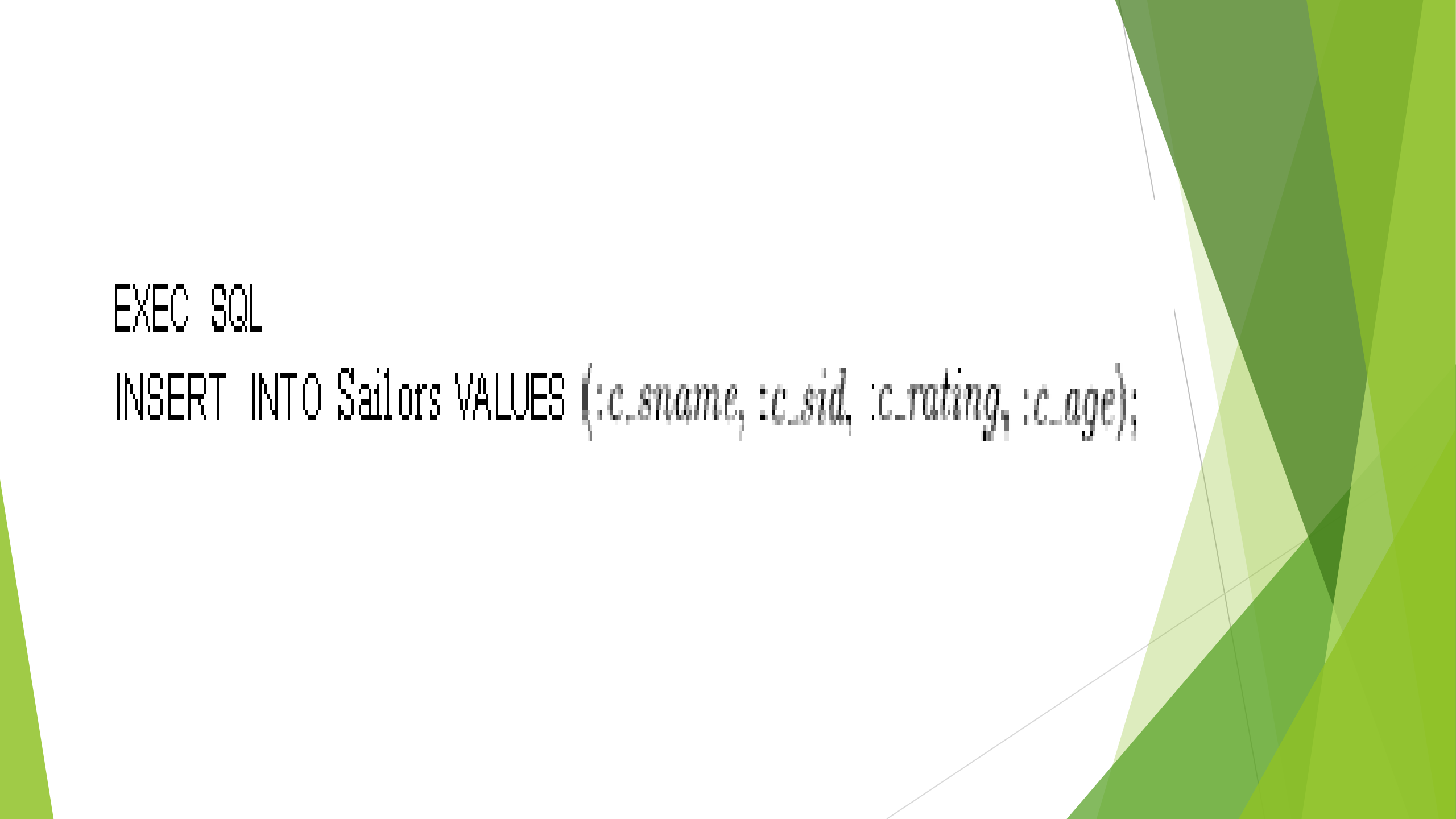
```
EXEC SQL BEGIN DECLARE SECTION  
char :c_sname[20];  
long :csid;  
short :crating;  
float :cage;  
EXEC SQL END DECLARE SECTION
```

SQLCODE and SQLSTATE

- ▶ To report what went wrong if an **error condition** arises when executing an SQL statement.
- ▶ The SQL-92 standard recognizes two special variables for reporting errors, SQLCODE and SQLSTATE.

- ▶ **SQLCODE** is the older of the two and is defined to return some negative value when an error condition arises, without specifying further just what error a particular negative integer denotes.
- ▶ **SQLSTATE**, introduced in the SQL-92 standard for the first time, associates predefined values with several common error conditions, thereby introducing some uniformity to how errors are reported.

- 
- ▶ All SQL statements embedded within a host program must be clearly marked, SQL statements must be prefixed by EXEC SQL.
 - ▶ Observe that a semicolon terminates the command, as per the convention for terminating statements in C.

The background features abstract, overlapping green geometric shapes in various shades of green, primarily on the right side of the slide.

EXEC SQL

INSERT INTO Sailors VALUES (:c_sname, :c_sid, :c_rating, :c_age);

- ▶ The SQLSTATE variable should be checked for errors and exceptions after each Embedded SQL statement. SQL provides the WHENEVER command to simplify this tedious task

```
EXEC SQL WHENEVER [SQLERROR | NOT FOUND] [ CONTINUE | GOTO st'mt ]
```

- ▶ If SQLERROR is specified and the value of SQLSTATE indicates an exception, control is transferred to *stmt*, which is presumably responsible for error and exception handling.

PL/SQL

- ▶ When SQL statements are issued from a remote application, the records in the result of the query need to be transferred from the database system back to the application.
- ▶ Once a stored procedure is registered with the database server, different users can re-use the stored procedure, eliminating duplication of efforts in writing SQL queries or application logic, and making code maintenance.
- ▶ In addition, application programmers do not need to know the database schema if we encapsulate all database access into stored procedures

PL/SQL

- ▶ PL/SQL is a procedural language extension to Structured Query Language (SQL).
- ▶ The purpose of PL/SQL is to combine database language and procedural programming language.
- ▶ The basic unit in PL/SQL is called a block and is made up of three parts: a declarative part, an executable part and an exception-building part.
- ▶ A PL/SQL program that is stored in a database in compiled form and can be called by name is referred to as a stored procedure

Differences between SQL and PL/SQL

SQL	PL/SQL
SQL is a single query that is used to perform DML and DDL operations.	PL/SQL is a block of codes that used to write the entire program blocks/ procedure/ function, etc.
Execute as a single statement.	Execute as a whole block.
Mainly used to manipulate data.	Mainly used to create an application.
Cannot contain PL/SQL code in it.	It is an extension of SQL, so it can contain SQL inside it.

Structure of PL/SQL Block

- ▶ The basic unit in PL/SQL is a block. All PL/SQL programs are made up of blocks, which can be nested within each other. PL/SQL contains 4 main blocks
 - ▶ Declare
 - ▶ Begin
 - ▶ Exception
 - ▶ End
- ▶ Typically, each block performs a logical action in the program. A block has the following structure:

```
DECLARE
    Declaration statement;
BEGIN
    Executable statements;

EXCEPTIONS
    Exception handling statements;

END;
```

PL/SQL Identifiers

- There are several PL/SQL identifiers such as **procedures, cursors, triggers** etc.

Variables:

- Like several other programming languages, variables in PL/SQL must be declared prior to its use. They should have a valid name and data type as well.

```
SQL> SET SERVEROUTPUT ON;
SQL> DECLARE
    var1 INTEGER;
    var2 REAL;
    var3 varchar2(20) ;
BEGIN
    null;
END;
/

Output:
PL/SQL procedure successfully completed
```


INITIALISING VARIABLES

- Example for initializing variable

```
SQL> SET SERVEROUTPUT ON;
SQL> DECLARE
    var1 INTEGER := 2 ;
    var3 varchar2(20) := 'Hi Everyone!!!!';

    BEGIN
        null;
    END;
    /
```

Output:
PL/SQL procedure successfully completed

DISPLAYING OUTPUT

- The outputs are displayed by using DBMS_OUTPUT which is a built-in package that enables the user to display output.

```
SQL> SET SERVEROUTPUT ON;  
SQL> DECLARE  
        var varchar2(40) := 'Hi Everyone';  
  
        BEGIN  
            dbms_output.put_line(var);  
  
        END;  
        /
```

Output:

```
Hi Everyone  
PL/SQL procedure successfully completed
```

TAKING INPUT FROM USER

- ▶ Just like in other programming languages, in PL/SQL also, we can take input from the user and store it in a variable

```
SQL> SET SERVEROUTPUT ON;  
SQL> DECLARE  
    -- taking input for variable a  
    a number := &a;  
    -- taking input for variable b  
    b varchar(30) := &b;  
BEGIN  
    null;  
END;  
/
```

Output:

PL/SQL procedure successfully completed

Example 1:

```
--PL/SQL code to print sum of two numbers taken from the user.  
SQL> SET SERVEROUTPUT ON;  
SQL> DECLARE  
    -- taking input for variable a  
    a integer := &a ;  
    -- taking input for variable b  
    b integer := &b ;  
    c integer ;  
BEGIN  
    c := a + b ;  
    dbms_output.put_line('Sum of ||a|| and ||b|| is = ||c||');  
END;  
/
```

Output:

2

3

Sum of 2 and 3 is =5

PL/SQL procedure successfully completed

Example 2:

--PL/SQL code to print natural numbers from 1 to 5

```
SQL> SET SERVEROUTPUT ON;
```

```
SQL> DECLARE
```

```
    i number;
```

```
BEGIN
```

```
    i:=1;
```

```
    loop
```

```
        dbms_output.put_line(i);
```

```
        i:= i+1;
```

```
        exit when i>5;
```

```
    end loop;
```

```
END;
```

```
/
```

Procedure (stored procedure)

- ▶ A stored procedure is a prepared SQL code that you can save, so the code can be reused over and over again.
- ▶ You can also pass parameters to a stored procedure. They are:

IN:

This is the Default Parameter for the procedure. It always receives the values from calling program.

OUT:

This parameter always sends the values to the calling program.

IN OUT:

This parameter performs both the operations. It Receives value from as well as sends the values to the calling program.

Syntax : Creating a Procedure

CREATE or REPLACE PROCEDURE name(parameters)

IS

variables;

BEGIN

//statements;

END;

```
CREATE PROCEDURE FINDMIN(X  
IN NUMBER,Y IN NUMBER,Z OUT  
NUMBER) IS  
BEGIN  
IF X<Y THEN  
Z:=X;  
ELSE  
Z:=Y;  
END IF;  
END;  
/
```

Procedure created.

```
SQL> DECLARE  
A NUMBER:=20;  
B NUMBER:=10;  
C NUMBER;  
BEGIN  
FINDMIN(A,B,C);  
DBMS_OUTPUT.PUT_LINE('MIN OF A AND B IS' || C);  
END;  
/
```

MIN OF A AND B IS10

PL/SQL procedure successfully completed.

CURSORS

- ▶ It essentially provide a mechanism that allows us to retrieve rows one at a time from a relation. This mechanism is called a cursor.
- ▶ Cursors are used when the user needs to update records in a singleton fashion or in a row by row manner, in a database table.
- ▶ A cursor can be declare on any relation or on any SQL query (because every query returns a set of rows).
- ▶ Once a cursor is declared,
 1. Open it (which positions the cursor just before the first row);
 2. Fetch the next row
 3. move the cursor
 4. Close the cursor

Basic Cursor Definition and Usage

Cursors enable us to examine, in the host language program, a collection of JWS computed by an Embedded SQL statement:

- We need to open a cursor if the embedded statement is a SELECT (i.e.) a query). INSERT, DELETE, and UPDATE statements typically require no cursor, although some variants of DELETE and UPDATE use a cursor.

As an example, we can find the name and age of a sailor, specified by assigning a value to the host variable `c_sid`, declared earlier, as follows:

```
EXEC SQL SELECT S.sname, S.age
        INTO   :c_sname, :c_age
        FROM   Sailors S
        WHERE  S.sid = :c_sid;
```

- The INTO clause allows us to assign the columns of the single answer row to the host variables `csname` and `c_age`. Therefore, we do not need a cursor to embed this query in a host language program. But for following query returns a collection of rows, not just one row.

```
SELECT S.sname, S.age  
FROM   Sailors S  
WHERE  S.rating > :c_minrating
```

- When executed interactively, the answers are printed on the screen. If we embed this query in a C program by prefixing the command with EXEC SQL, The INTO clause is inadequate because we must deal with several rows. The solution is to use a cursor:

```
DECLARE sinfo CURSOR FOR  
SELECT S.sname, S.age  
FROM   Sailors S  
WHERE  S.rating > :c_minrating;
```

- This code can be included in a C program, and once it is executed, the cursor sinfo is defined. Subsequently, we can open the cursor:

OPEN *sinfo*;

The value of *c_minrating* in the SQL query associated with the cursor is the value of this variable when we open the cursor. A cursor can be thought of as 'pointing' to a row in the collection of answers to the query associated with it. When a cursor is opened, it is positioned just before the first row. We can use the **FETCH** command to read the first row of cursor *sinfo* into host language variables:

FETCH *sinfo* **INTO** :c_sname, :c_age;

When the **FETCH** statement is executed, the cursor is positioned to point at the next row (which is the first row in the table when **FETCH** is executed for the first time after opening the cursor) and the column values in the row are copied into the corresponding host variables. By repeatedly executing this **FETCH** statement, we can read all the rows computed by the query, one row at a time. Additional parameters to the **FETCH** command allow us to position a cursor in very flexible ways.

When we are done with a cursor, we can close it:

CLOSE *sinfo*;

Properties of Cursors

The general form of a cursor declaration is:

```
DECLARE cursorname [INSENSITIVE] [SCROLL] CURSOR  
      [WITH HOLD]  
      FOR some query  
      [ORDER BY order-item-list ]  
      [FOR READ ONLY | FOR UPDATE ]
```

- i) A cursor can be declared to be a read-only cursor (FOR READ ONLY) or, if it is a cursor on a base relation or an updatable view, to be an updatable cursor (FOR UPDATE).
- ii) If it is Updatable, simple variants of the UPDATE and DELETE commands allow us to update or delete the row on which the cursor is positioned. For example, if *sinfo* is an updatable cursor and open, we can execute the following statement:

```
UPDATE Sailors S  
SET      S.rating = S.rating - 1  
WHERE    CURRENT of sinfo;
```


This Embedded SQL statement modifies the *rating* value of the row currently pointed to by cursor *sinfo*; similarly, we can delete this row by executing the next statement:

```
DELETE Sailors S  
WHERE CURRENT of sinfo;
```

- iii) A cursor is updatable by default unless it is a scrollable or insensitive cursor (see below), in which case it is read-only by default.
- iv) If the keyword SCROLL is specified, the cursor is scrollable, which means that variants of the FETCH command can be used to position the cursor in very flexible ways; otherwise, only the basic FETCH command, which retrieves the next row, is allowed.
- v) If the keyword INSENSITIVE is specified, the cursor behaves as if it restricts actions of some transaction that could modify these rows, creating unpredictable behaviour.
- vi) A holdable cursor is specified using the WITH HOLD clause. The motivation for this comes from long transactions in which we access a large number of rows of a table. If the transaction is aborted for any reason, the system potentially has to redo a lot of work when the transaction is restarted. Even if the transaction is not aborted, its locks are held for a long time and reduce the concurrency of the system. The alternative is to break the transaction into several smaller transactions, but remembering our position in the table between transactions

- vii) The order in which FETCH commands retrieve is unspecified, but the optional ORDER BY clause can be used to specify a sort order.

The order-item-list is a list of order-items; an order-item is a column name, optionally followed by one of the keywords ASC or DESC. The keywords ASC or DESC that follow a column control whether the result should be sorted-with respect to that column-in ascending or descending order; the default is ASC. This clause is applied as the last step in evaluating the query.

Consider the query discussed in Section 5.5.1, and the answer shown in Figure 5.13. Suppose that a cursor is opened on this query, with the clause:

ORDER BY minage ASC, rating DESC

The answer is sorted first in ascending order by *minage*, and if several rows have the same *minage* value, these rows are sorted further in descending order by *rating*. The cursor would fetch the rows in the order shown in Figure 6.1.

<i>rating</i>	<i>minage</i>
8	25.5
3	25.5
7	35.0

Figure 6.1 Order in which Tuples Are Fetched

Employee11 table:

EMP_NO	E_NAME	E_ADDRESS	E_PH_NO	DEPT_NO	JOB_ID DESIGNATION	SALARY
10	ASHOK	BANGALORE	9087234567	100	ASSISTANT	10000
20	RAMAN	MYSORE	989898989	200	ASSOCIATE	20000
30	SHWETA	TUMKUR	9772341345	300	PROFESSOR	30000
40	RAHUL	VIZAG	8907654321	400	MANAGER	40000
50	BABU	HYD	9087878989	500	MANAGER	50000


```
SQL> SET SERVEROUTPUT ON;
SQL> DECLARE
  2  CURSOR SAMPLE1 IS SELECT * FROM EMPLOYEE11 WHERE DESIGNATION='MANAGER';
  3  SAMPLE_R1 SAMPLE1%ROWTYPE;
  4  BEGIN
  5  OPEN SAMPLE1;
  6  LOOP
  7  FETCH SAMPLE1 INTO SAMPLE_R1;
  8  EXIT WHEN SAMPLE1%NOTFOUND;
  9  DBMS_OUTPUT.PUT_LINE(SAMPLE_R1.EMP_NO||' '||SAMPLE_R1.E_NAME||' '||SAMPLE_R1.SALARY);
10  END LOOP;
11  END;
12  /
40 RAHUL 40000
50 BABU 50000
```

PL/SQL procedure successfully completed.

```

SQL> SET SERVEROUTPUT ON;
SQL> DECLARE
  2   CURSOR SAMPLE2 IS SELECT * FROM EMPLOYEE11 WHERE SALARY>any (SELECT SALARY FROM EMPLOYEE11 WHERE DESIGNATION='IT STAFF');
  3   SAMPLE_R2 SAMPLE2%ROWTYPE;
  4   BEGIN
  5   OPEN SAMPLE2;
  6   LOOP
  7   FETCH SAMPLE2 INTO SAMPLE_R2;
  8   EXIT WHEN SAMPLE2%NOTFOUND;
  9   DBMS_OUTPUT.PUT_LINE(SAMPLE_R2.EMP_NO||' '||SAMPLE_R2.E_NAME||' '||SAMPLE_R2.E_ADDRESS||' '||SAMPLE_R2.E_PH_NO||' '||SAMPLE_R2.DEPT_NO||' '||SAMP
LE_R2.DEPT_NAME||' '||SAMPLE_R2.JOB_ID||' '||SAMPLE_R2.DESIGNATION||' '||SAMPLE_R2.SALARY);
 10   END LOOP;
 11   END;
 12   /
50 BABU HYD 9087878989 50 HR 500 MANAGER 50000
40 RAHUL VIZAG 8907654321 40 CIVIL 400 MANAGER 40000
30 SHWETA TUMKUR 9772341345 30 MECH 300 PROFESSOR 30000
20 RAMAN MYSORE 9898989898 20 ECE 200 ASSOCIATE 20000
80 hegde kodagu 8907623451 80 ECE 800 IT STAFF 18000
70 hashmi kolar 8907623451 70 CSE 700 IT STAFF 15000
60 hazarika hyd 8907623451 60 CSE 600 IT STAFF 10000
10 ASHOK BANGALORE 9087234567 10 CSE 100 ASSISTANT 10000

```

PL/SQL procedure successfully completed.

ODBC and JDBC

- **Embedded SQL enables the integration of SQL with a general-purpose programming language.** A DBMS-specific pre-processor transforms the Embedded SQL statements into function calls in the host language.
- **ODBC and JDBC, short for Open Database Connectivity and Java Database Connectivity, also enable the integration of SQL with a general-purpose programming language.**
- Both ODBC and JDBC expose database capabilities in a standardized way to the application programmer through an application programming interface (API).
- Using ODBC or JDBC, an application can access not just one DBMS but several different ones simultaneously.
- **All direct interaction with a specific DBMS happens through a DBMS-specific driver. A driver is a software program that translates the ODBC or JDBC calls into DBMS-specific calls.**

- Drivers are loaded dynamically on demand since the DBMSs the application is going to access are known only at run-time.
- It is sufficient that the **driver translates the SQL commands from the application into equivalent commands that the DBMS understands.**
- An application that interacts with a data source through ODBC or JDBC selects a data source, dynamically loads the corresponding driver, and establishes a connection with the data source.
- **While a connection is open, transactions are executed by submitting SQL statements, retrieving results, processing errors, and finally committing or rolling back. The application disconnects from the data source to terminate the interaction.**

Architecture

- ▶ The architecture of JDBC has four main components: *the application, the corresponding data sources, the driver manager and several data source specific drivers,*
- i) The *application* initiates and terminates the **connection with a data source**. It submits SQL statements, and retrieves the results-----all through a well-defined interface as specified by the JDBC API.
- ii) The *data source* **processes commands from the driver** and returns the results. Depending on the relative location of the data source and the application, several architectural scenarios are possible.
- iii) The primary goal of the *driver manager* is to **load JDBC drivers and pass JDBC function calls from the application to the correct driver.**
- iv) The **data source specific driver** handles **JDBC initialization and information calls from the applications and can log all function calls**. In addition, it performs **error checking & establishes the connection with the data source**. In addition to submitting requests and returning request results, the driver translates data, error formats, and error codes from a form that is specific to the data source into the JDBC standard.

Definition of JDBC

- ▶ It is a standard interface between any Java application and different databases. The **function of JDBC is to help the Java-based application to access different types of databases.** JDBC provide methods to query database, and it can also be used to update the database. JDBC provide **JDBC drivers** that converts the request from Java application on client side to the language that database understands.
- ▶ As JDBC is language and platform specific, Java application can use **JDBC-to-ODBC Bridge** to communicate with ODBC adaptable databases. Unlike ODBC, JDBC has easy coding but, it is only limited to Java only.

JDBC

- ▶ Java database connectivity. Its an API.
- ▶ It enable Java applications to use SQL for database access.
- ▶ Its acts like an interface between java program and database.
- ▶ It pass the value from frontend to backend.

Procedure :

Import the package , register the jdbc driver , open a connection ,
Execute a query , extract data from resultset, close connection.

Definition of ODBC

- ▶ ODBC is **Open Database Connectivity**. Like JDBC, ODBC is also an API that acts as an interface between an application on the client side and the database on the server side.
- ODBC helps an application to access the data from the database. An application written in any language can use ODBC to access different types of databases and hence, it is said to be language and platform independent.
- ODBC is most widely used and understands many different programming languages. But its code is complex and hard to understand.
- ▶ **Similarity:** Both are used by the client-side applications to access different kinds of databases on server side.

Difference between ODBC and JDBC

ODBC	JDBC
ODBC Stands for Open Database Connectivity.	JDBC Stands for java database connectivity.
Introduced by Microsoft in 1992.	Introduced by SUN Micro Systems in 1997.
We can use ODBC for any language like C,C++,Java etc.	We can use JDBC only for Java languages.
We can choose ODBC only windows platform.	We can Use JDBC in any platform.
Mostly ODBC Driver developed in native languages like C,C++.	JDBC Stands for java database connectivity.
For Java applications it is not recommended to use ODBC because performance will be down due to internal conversion and applications will become platform Dependent.	For Java application it is highly recommended to use JDBC because there we no performance & platform dependent problem.

TRIGGERS

- ❑ A **trigger** is a procedure that is automatically invoked by the DBMS in response to specified changes to the database, and is typically specified by the DBA.
- ❑ A database that has a set of associated triggers is called an active database.
- ❑ A trigger description contains three parts:
 - ✓ **Event:** A **change to the database** that activates the trigger.
 - ✓ **Condition:** A query or **test** that is run when the trigger is activated.
 - ✓ **Action:** A **procedure that is executed** when the trigger is activated and its condition is true.

TRIGGERS

- ❑ An insert, delete or update statement could activate a trigger, regardless of which user or application invoked the activating statement.
- ❑ A **condition** in a trigger can be a true/false statement or a query.
- ❑ If the condition part evaluates to true, the action associated with the trigger is executed.

TRIGGERS

- ❑ In fact, an action can even execute a series of data-definition commands (e.g., create new tables, change authorizations) and transaction-oriented commands (e.g., commit), or call host language procedures.
- ❑ The execution of the **action part of a trigger could again activate the same trigger** such triggers are called recursive triggers.

TRIGGERS

Constraints versus Triggers

- ❑ **Triggers are active, while constraints are passive.** While constraints prevent updates that violate referential integrity, triggers perform explicit actions in addition to the update operation.
- ❑ Triggers can do much more than enforce referential integrity. Because they are passive, constraints are limited to preventing updates in a narrow set of conditions.
- ❑ A common use of triggers is to maintain database **consistency**.
- ❑ The meaning of a constraint is not defined operationally, unlike the effect of a trigger.
- ❑ This property makes a constraint easier to understand, and also gives the DBMS more opportunities to optimize execution.

TRIGGERS

Constraints versus Triggers

- ❑ A constraint also prevents the data from being made inconsistent by any kind of statement, whereas a trigger is activated by a specific kind of statement (e.g., an insert or delete statement). Again, this restriction makes a constraint easier to understand.
- ❑ On the other hand, triggers allow us to maintain database integrity in more flexible ways.

DESIGNING ACTIVE DATABASES

- ▶ Triggers offer a powerful mechanism for dealing with changes to a database, but they must be used with caution.
- ▶ The effect of a collection of triggers can be very complex and maintaining an active database can become very difficult.
- ▶ Often, a judicious use of integrity constraints can replace the use of triggers.

Why Triggers Can Be Hard to Understand

- ▶ In an active database system, when the DBMS is about to execute a statement that modifies the database, it checks whether some trigger is activated by the statement.
- ▶ If so, the DBMS processes the trigger by evaluating its condition part, and then (if the condition evaluates to true) executing its action part.
- ▶ If a statement activates more than one trigger, the DBMS typically processes all of them, in some arbitrary order. An important point is that the execution of the action part of a trigger could in turn activate another trigger.
- ▶ In particular, the execution of the action part of a trigger could again activate the same trigger; such triggers are called **recursive triggers**. The potential for such **chain activations, and the unpredictable order in which a DBMS processes activated triggers, can make it difficult to understand the effect of a collection of triggers.**

Other uses of Triggers

- ▶ A common use of triggers is to maintain database consistency.
- ▶ A constraint also prevents the data from being made inconsistent.
- ▶ Anyways, Triggers **allow us to maintain database integrity in more flexible ways.**
- ▶ **Triggers can alert users to unusual events** (as reflected in updates to the database).

For example, we may want to check whether a customer placing an order has made enough purchases in the past month to qualify for an additional discount; if so, the sales clerk must be informed so that he can tell the customer, and possibly generate additional sales!

- ▶ Triggers can generate a log of events to support auditing and security checks.

THANK YOU

Example:

Delimiter \$\$

Create procedure pro_cursor(id int) 101

Begin

declare name varchar(20);

declare mark int;

declare curl cursor for select stu_name from
stu_table where stu_id = id;

Open curl;

Fetch curl into name, mark;

Select name;

Close curl;

End\$\$

Output:

call pro_cursor(101);

TRIGGERS

Trigger Declaration Syntax

```
CREATE [OR REPLACE ] TRIGGER trigger_name
{ BEFORE | AFTER | INSTEAD OF }
{ INSERT [OR] | UPDATE [OR] | DELETE }
[OF col_name]
ON table_name
[REFERENCING OLD AS o NEW AS n]
[FOR EACH ROW]
WHEN (condition)
DECLARE
    Declaration-statements
BEGIN
    Executable-statements
EXCEPTION
    Exception-handling-statements
END;
```

Triggers Example

```
Create table student(  
  Studentid int primary key,  
  Fname varchar(20),  
  Lname varchar(20),  
  Address varchar(30),  
  City varchar(15),  
  Marks int);
```

BEFORE INSERT

```
delimiter $$  
Create Trigger calculate BEFORE INSERT ON student  
for each row  
Begin  
    set new.Marks= new.Marks+100;  
end;  
$$
```

Triggers Example

```
create table final_marks(  
Per int);
```

AFTER INSERT

```
delimiter $$  
Create Trigger total_marks AFTER INSERT ON student  
for each row  
Begin  
    insert into final_marks values(new.Marks);  
end;  
$$
```

Output:

```
Insert into student values(1,'raj','k','an','blore',78);  
  
select * from final_marks;
```

Triggers Example: BEFORE UPDATE

```
create table student_marks(  
  Studentid int primary key,  
  Name varchar(20),  
  Sub1 int,  
  Sub2 int,  
  Sub3 int,  
  Total int,  
  Per_marks int);  
  
DELIMITER $$  
CREATE TRIGGER `student_marks_upd`  
BEFORE UPDATE ON student_marks  
FOR EACH ROW  
BEGIN  
  SET new.Total = new.Sub1 + new.Sub2 + new.Sub3 ;  
  SET new.Per_marks = new.Total/3;  
end;  
$$
```

Output:

```
Update student_marks set Sub1 = 54, Sub2 = 69, Sub3 = 89;  
Select * from student_marks;
```

Trigger Example: AFTER UPDATE

Delimiter \$\$

Create trigger stu_update

AFTER UPDATE ON stu_table

For each row

Begin

```
insert into stu_log values(user(), concat('update student
record',('old.stu_id' , 'old.stu_name') to
```

```
( ' , new.stu_id', ' ,new.stu_name')));
```

End;

\$\$

stu_table

stu_id	stu_name
--------	----------

stu_log

user	description
------	-------------