1)a) To retrieve the details of all books in the Book_id, title, name of the publisher, and authors, you can use the following SQL query:

```sql
SELECT b.Book_id, b.Title, p.Name AS Publisher_Name, ba.Author_Name
FROM BOOK b
JOIN BOOK_AUTHORS ba ON b.Book_id = ba.Book_id
JOIN PUBLISHER p ON b.Publisher_Name = p.Name;
```

b) To get the particulars of borrowers who have borrowed more than 3 books from January 2017 to June 2017, you can use the following SQL query:

```sql
SELECT *
FROM Borrower
WHERE Borrower_id IN (
    SELECT Borrower_id
    FROM Borrowed_Books
    WHERE Borrow_Date BETWEEN '2017-01-01' AND '2017-06-30'
    GROUP BY Borrower_id
    HAVING COUNT(*) > 3
);
```

Note: The schema does not include the Borrower and Borrowed_Books tables, so I assumed their existence for this query.

c) To delete a book from the BOOK table and update the related tables, you can use the following SQL statements:

```sql
```

```
-- First, delete the book from the BOOK table
DELETE FROM BOOK
WHERE Book_id = <book_id_to_delete>;

-- Then, delete the corresponding entries from the BOOK_AUTHORS table
DELETE FROM BOOK_AUTHORS
WHERE Book_id = <book_id_to_delete>;
```

d) To create a view of all books and their number of copies that are currently available in the library, you can use the following SQL statement:

```sql
CREATE VIEW Available_Books AS
SELECT b.Book_id, b.Title, b.No_of_copies
FROM BOOK b
WHERE b.No_of_copies > 0;
```

e) Here's an example of a PL/SQL procedure to display the book details of a particular author:

```sql
CREATE OR REPLACE PROCEDURE GetBookDetailsByAuthor(
  author_name IN VARCHAR2
)
IS
BEGIN
  FOR book_rec IN (
    SELECT b.Book_id, b.Title, b.Publisher_Name
    FROM BOOK b
```

```
    JOIN BOOK_AUTHORS ba ON b.Book_id = ba.Book_id
    WHERE ba.Author_Name = author_name
  )
  LOOP
    DBMS_OUTPUT.PUT_LINE('Book ID: ' || book_rec.Book_id);
    DBMS_OUTPUT.PUT_LINE('Title: ' || book_rec.Title);
    DBMS_OUTPUT.PUT_LINE('Publisher: ' || book_rec.Publisher_Name);
    DBMS_OUTPUT.PUT_LINE('---');
  END LOOP;
END;
```

You can call this PL/SQL procedure by passing the author's name as the input parameter:

```sql
BEGIN
  GetBookDetailsByAuthor('John Smith');
END;
```

Replace 'John Smith' with the actual author's name you want to search for. The procedure will display the book details for that author using the DBMS_OUTPUT.PUT_LINE statement. Remember to enable DBMS_OUTPUT in your PL/SQL environment to see the output.

---

2)

a) To alter the table by adding a column named "salary," you can use the following SQL statement:

```sql
ALTER TABLE employee
```

ADD salary DECIMAL(10, 2);

```

b) To copy the table "employee" as "Emp," you can use the CREATE TABLE AS SELECT (CTAS) statement:

```sql
CREATE TABLE Emp AS
SELECT *
FROM employee;
```

c) To delete the 2nd row from the table, you need a primary key or a unique identifier column. Let's assume the "S.No" column serves as the primary key. You can use the following SQL statement to delete the 2nd row:

```sql
DELETE FROM employee
WHERE "S.No" = 2;
```

d) To drop the table "employee," you can use the DROP TABLE statement:

```sql
DROP TABLE employee;
```
```

e) To demonstrate triggers for automatic updation, let's consider an example where we want to update the "salary" column automatically whenever the "Desination" column is updated. Here's how you can achieve this:

```sql
-- Create the employee table
CREATE TABLE employee (
  "S.No" INT,
  Name VARCHAR(50),
  Desination VARCHAR(50),
  Branch VARCHAR(50),
  Salary DECIMAL(10, 2)
);

-- Create the trigger
CREATE OR REPLACE TRIGGER salary_update_trigger
BEFORE UPDATE OF Desination ON employee
FOR EACH ROW
BEGIN
  -- Update the salary column to a new value based on the updated Desination
  IF :NEW.Desination = 'Manager' THEN
    :NEW.Salary := :NEW.Salary * 1.1; -- Increase the salary by 10% for managers
  ELSIF :NEW.Desination = 'Supervisor' THEN
    :NEW.Salary := :NEW.Salary * 1.05; -- Increase the salary by 5% for supervisors
  END IF;
END;
```

/

```
---

3)

a) To display the Emp_name who belongs to Emp_dept = "xxx" along with salary using the GROUP BY clause, you can use the following SQL query:

```sql
SELECT Emp_name, Sal
FROM Employee
WHERE Emp_dept = 'xxx'
GROUP BY Emp_name, Sal;
```

Replace 'xxx' with the actual department name.

b) To display the lowest paid employee details under each department, you can use the following SQL query:

```sql
SELECT Emp_dept, Emp_name, Sal
FROM Employee
WHERE (Emp_dept, Sal) IN (
  SELECT Emp_dept, MIN(Sal)
  FROM Employee
  GROUP BY Emp_dept
);
```
```

c) To list the employee names in descending order, you can use the following SQL query:

```sql
SELECT Emp_name
FROM Employee
ORDER BY Emp_name DESC;
```

d) To rename the column of the Employee table using the ALTER command, you can use the following SQL statement:

```sql
ALTER TABLE Employee
RENAME COLUMN Emp_name TO Employee_name;
```

Replace "Emp_name" with the actual column name you want to rename, and "Employee_name" with the new column name.

e) To insert a row in the Employee table using triggers, you need to create an appropriate trigger that specifies the insertion conditions and actions. Here's an example:

```sql
-- Create the Employee table
CREATE TABLE Employee (
```

```
  Emp_no INT,
  Emp_name VARCHAR(50),
  Emp_dept VARCHAR(50),
  Job VARCHAR(50),
  Mgr VARCHAR(50),
  Sal DECIMAL(10, 2)
);
-- Create the trigger
CREATE OR REPLACE TRIGGER insert_employee_trigger
BEFORE INSERT ON Employee
FOR EACH ROW
BEGIN
  -- Perform some action or validation before insertion
  -- Example: Enforce a minimum salary of 1000
  IF :NEW.Sal < 1000 THEN
    RAISE_APPLICATION_ERROR(-20001, 'Salary must be at least 1000.');
  END IF;
END;
/
```

The trigger above will execute before every insert operation on the "Employee" table. It checks if the salary is less than 1000 and raises an error if the condition is not met.

---

4) a) To grant some privileges of the "Employees" table into the "Departments" table, you can use the GRANT statement. Here's an example:

```sql
GRANT SELECT, INSERT, UPDATE ON Employees TO Departments;
```

```

This grants the SELECT, INSERT, and UPDATE privileges on the "Employees" table to the "Departments" table. Adjust the privileges and table names as per your requirements.

b) To revoke all privileges of the "Employees" table from the "Departments" table, you can use the REVOKE statement. Here's an example:

**```sql**

**REVOKE ALL PRIVILEGES ON Employees FROM Departments;**

```

This revokes all privileges on the "Employees" table from the "Departments" table.

c) To revoke some privileges of the "Employees" table from the "Departments" table, you can use the REVOKE statement with specific privileges. Here's an example:

```sql

REVOKE SELECT, UPDATE ON Employees FROM Departments;

```

This revokes the SELECT and UPDATE privileges on the "Employees" table from the "Departments" table. Adjust the privileges and table names as per your requirements.

d) To implement a savepoint in a transaction, you can use the SAVEPOINT statement. Here's an example:
```

```sql
SAVEPOINT my_savepoint;
```

This creates a savepoint named "my_savepoint" in the current transaction. You can use this savepoint to roll back to a specific point within the transaction if needed.

To roll back to the savepoint, you can use the ROLLBACK TO SAVEPOINT statement:

```sql
ROLLBACK TO SAVEPOINT my_savepoint;
```

This rolls back the transaction to the specified savepoint, undoing any changes made after the savepoint.

e) Here's an example of a user-defined procedure for the employee database:

```sql
CREATE OR REPLACE PROCEDURE GetEmployeeDetails(
  emp_id IN NUMBER
)
IS
  emp_name VARCHAR2(50);
  emp_salary NUMBER;
  dept_name VARCHAR2(50);
BEGIN
```

```
  -- Fetch employee details based on emp_id
  SELECT emp_name, emp_salary, dept_name
  INTO emp_name, emp_salary, dept_name
  FROM Employees e
  JOIN Departments d ON e.dept_no = d.dept_no
  WHERE e.emp_id = emp_id;

  -- Display the employee details
  DBMS_OUTPUT.PUT_LINE('Employee ID: ' || emp_id);
  DBMS_OUTPUT.PUT_LINE('Employee Name: ' || emp_name);
  DBMS_OUTPUT.PUT_LINE('Employee Salary: ' || emp_salary);
  DBMS_OUTPUT.PUT_LINE('Department Name: ' || dept_name);
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('Employee not found.');
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Error occurred: ' || SQLERRM);
END;
/
```

You can call this procedure by passing the employee ID as an input parameter:

```sql
BEGIN
  GetEmployeeDetails(1001);
END;
```

Replace 1001 with the actual employee ID you want to retrieve the details for. The procedure will display the employee's name, salary, and department using the DBMS_OUTPUT.PUT_LINE statements.

---

5) a) Here are the appropriate primary keys and foreign keys for the given tables:

- Event:
  - Primary Key: eventid

- Participant:
  - Primary Key: playerid
  - Foreign Key: eventid (references Event(eventid))

- Prizes:
  - Primary Key: prizeid
  - Foreign Key: eventid (references Event(eventid))

- Winners:
  - Foreign Key: prizeid (references Prizes(prizeid))
  - Foreign Key: playerid (references Participant(playerid))

b) To ensure that the playerid contains at least one digit character, you can add a CHECK constraint to the Participant table. Here's an example:

```sql
ALTER TABLE Participant
ADD CONSTRAINT chk_playerid CHECK (REGEXP_LIKE(playerid, '\d'));
```

This constraint ensures that the playerid column must contain at least one digit character.

c) To retrieve the name of events where all prize winners are females, you can use the following SQL query:

```sql
SELECT e.name AS event_name
FROM Event e
WHERE NOT EXISTS (
  SELECT *
  FROM Prizes p
  JOIN Winners w ON p.prizeid = w.prizeid
  JOIN Participant pt ON w.playerid = pt.playerid
  WHERE e.eventid = p.eventid
    AND pt.gender != 'Female'
);
```

This query uses a subquery to check if there are any prize winners who are not females for each event. The events where all prize winners are females will be retrieved.

d) To create a non-updatable view to retrieve the names of all participants who won 1st prizes along with their event names, you can use the following SQL statement:

```sql
```

CREATE VIEW FirstPrizeWinners AS

SELECT p.name AS participant_name, e.name AS event_name

FROM Participants p

JOIN Winners w ON p.playerid = w.playerid

JOIN Prizes pz ON w.prizeid = pz.prizeid

JOIN Event e ON pz.eventid = e.eventid

WHERE pz.rank = 1;

```

This view combines the Participant, Winners, Prizes, and Event tables to retrieve the names of participants who won the 1st prize along with their respective event names.

e) To write a trigger to ensure that for every new event created, 3 prizes are created in the prizes table, you can use the following SQL statement:

**```sql**

**CREATE OR REPLACE TRIGGER CreateEventPrizes**

**AFTER INSERT ON Event**

**FOR EACH ROW**

**BEGIN**

**  INSERT INTO Prizes (prizeid, prize_money, eventid, rank)**

**  VALUES (NEW.eventid || '_1', 1500, NEW.eventid, 1);**

**  INSERT INTO Prizes (prizeid, prize_money, eventid, rank)**

**  VALUES (NEW.eventid || '_2', 1000, NEW.eventid, 2);**

**  INSERT INTO Prizes (prizeid, prize_money, eventid, rank)**

**  VALUES (NEW.eventid || '_3', 500, NEW.eventid, 3);**

**END;**
```

/

```

This trigger fires after every insert operation on the Event table. It automatically inserts three prize records into the Prizes table for the newly created event. The prizeid is generated based on the eventid and rank, and the prize_money values are set accordingly.

---

6) a) To list the titles of all movies directed by 'XXXX', you can use the following SQL query:

```sql
SELECT Mov_Title
FROM MOVIES
JOIN DIRECTOR ON MOVIES.Dir_id = DIRECTOR.Dir_id
WHERE Dir_Name = 'XXXX';
```

Replace 'XXXX' with the actual name of the director.

b) To find the movie names where one or more actors acted in two or more movies, you can use the following SQL query:

```sql
SELECT Mov_Title
FROM MOVIES
JOIN MOVIE_CAST ON MOVIES.Mov_id = MOVIE_CAST.Mov_id
GROUP BY Mov_Title
HAVING COUNT(DISTINCT MOVIE_CAST.Act_id) >= 2;
```
```

```

This query joins the MOVIES and MOVIE_CAST tables and groups the results by movie title. It then uses the HAVING clause to filter for movies where the count of distinct actor IDs is greater than or equal to 2.

c) To list all actors who acted in a movie before 2010 and also in a movie after 2015 using a JOIN operation, you can use the following SQL query:

```sql
SELECT a.Act_Name
FROM ACTOR a
JOIN MOVIE_CAST mc1 ON a.Act_id = mc1.Act_id
JOIN MOVIES m1 ON mc1.Mov_id = m1.Mov_id
JOIN MOVIE_CAST mc2 ON a.Act_id = mc2.Act_id
JOIN MOVIES m2 ON mc2.Mov_id = m2.Mov_id
WHERE m1.Mov_Year < 2010 AND m2.Mov_Year > 2015;
```

This query joins the ACTOR, MOVIE_CAST, and MOVIES tables multiple times to match actors who acted in movies before 2010 and after 2015.

d) To create a view of movies with a particular actor and director, you can use the following SQL statement:

```sql
CREATE VIEW MovieActorDirector AS
SELECT MOVIES.Mov_Title, ACTOR.Act_Name, DIRECTOR.Dir_Name
```
```

```
FROM MOVIES

JOIN MOVIE_CAST ON MOVIES.Mov_id = MOVIE_CAST.Mov_id

JOIN ACTOR ON MOVIE_CAST.Act_id = ACTOR.Act_id

JOIN DIRECTOR ON MOVIES.Dir_id = DIRECTOR.Dir_id;
```

This view combines the MOVIES, MOVIE_CAST, ACTOR, and DIRECTOR tables to retrieve the movie title, actor name, and director name.

e) Here's an example of a user-defined function for the movie database:

```sql
CREATE OR REPLACE FUNCTION GetMovieCountForActor(actor_name IN VARCHAR2)

RETURN NUMBER

IS

  movie_count NUMBER;

BEGIN

  SELECT COUNT(*) INTO movie_count

  FROM MOVIE_CAST mc

  JOIN ACTOR a ON mc.Act_id = a.Act_id

  WHERE a.Act_Name = actor_name;


  RETURN movie_count;

END;

/
```

This function takes an actor name as input and returns the count of movies in which the actor has acted. You can call this function as follows:

```sql
DECLARE
  actor_name VARCHAR2(50) := 'John Doe';
  count NUMBER;
BEGIN
  count := GetMovieCountForActor(actor_name);
  DBMS_OUTPUT.PUT_LINE(actor_name || ' has acted in ' || count || ' movies.');
END;
/
```

Replace 'John Doe' with the actual actor name you want to retrieve the movie count for. The function will return the count of movies and display it using the DBMS_OUTPUT.PUT_LINE statement.

---

7) a) To compute the total number of male and female students in each semester and in each section, you can use the following SQL query:

```sql
SELECT Sem, Gender, COUNT(*) AS Total_Students
FROM STUDENT
GROUP BY Sem, Gender;
```

This query groups the students by semester and gender, and then counts the number of students in each group.

b) To calculate the Finalmark (average of the best two test marks) and update the corresponding table for all students, you can use the following SQL statement:

```sql
UPDATE MARKS m
SET Finalmark = (m.Test1 + m.Test2 + m.Test3 - LEAST(m.Test1, m.Test2, m.Test3)) / 2;
```

This query updates the Finalmark column in the MARKS table for all students. It calculates the average of the best two test marks (excluding the lowest mark) using the LEAST function.

c) To categorize students based on the Finalmark criterion, you can use the following SQL query:

```sql
UPDATE MARKS
SET CAT = CASE
    WHEN Finalmark BETWEEN 81 AND 100 THEN 'Outstanding'
    WHEN Finalmark BETWEEN 51 AND 80 THEN 'Average'
    WHEN Finalmark < 51 THEN 'Weak'
    ELSE NULL -- Handle any other cases
  END;
```

This query updates the CAT column in the MARKS table based on the Finalmark values using a CASE statement.

d) To create a view of Test3 marks of a particular student in all subjects, you can use the following SQL statement:

```sql
CREATE VIEW StudentTest3Marks AS
SELECT RegNo, Subcode, Test3
FROM MARKS;
```

This view retrieves the RegNo, Subcode, and Test3 columns from the MARKS table.

e) Here's an example of a stored procedure to demonstrate the above operations for the College Database:

```sql
CREATE OR REPLACE PROCEDURE UpdateStudentMarksAndCategorize
IS
BEGIN
  -- Calculate Finalmark and update the MARKS table
  UPDATE MARKS m
  SET Finalmark = (m.Test1 + m.Test2 + m.Test3 - LEAST(m.Test1, m.Test2, m.Test3)) / 2;

  -- Categorize students based on Finalmark
  UPDATE MARKS
```

```
  SET CAT = CASE
     WHEN Finalmark BETWEEN 81 AND 100 THEN 'Outstanding'
     WHEN Finalmark BETWEEN 51 AND 80 THEN 'Average'
     WHEN Finalmark < 51 THEN 'Weak'
     ELSE NULL -- Handle any other cases
   END;
END;
/
```

This procedure calculates the Finalmark for all students and updates the MARKS table. It also categorizes the students based on the Finalmark values using a CASE statement.

You can execute the procedure using the following command:

```sql
BEGIN
  UpdateStudentMarksAndCategorize;
  DBMS_OUTPUT.PUT_LINE('Marks updated and students categorized successfully.');
END;
/
```

---

8) a) To select rows from the Bank table using a WHERE clause, you can use the following SQL query:

```sql
```

SELECT *

FROM Bank

WHERE Branch = 'XYZ';

```

Replace 'XYZ' with the desired branch name. This query selects all columns from the Bank table where the Branch column is equal to 'XYZ'.

b) To select rows from the Bank table using a comparison operator, you can use the following SQL query:

```sql

**SELECT ***

**FROM Bank**

**WHERE Balance > 5000;**

```

This query selects all columns from the Bank table where the Balance column is greater than 5000. You can modify the comparison operator and value as per your requirement.

c) To update the balance in the second row of the Bank table, you can use the following SQL statement:

```sql

**UPDATE Bank**

**SET Balance = 1500**

**WHERE S.No = 2;**

```

This query updates the Balance column of the Bank table to 1500 where the S.No is equal to 2. Modify the value and condition according to your specific requirement.

d) To select rows from the Bank table using the BETWEEN operator on the Balance field, you can use the following SQL query:

```sql
SELECT *
FROM Bank
WHERE Balance BETWEEN 1000 AND 5000;
```

This query selects all columns from the Bank table where the Balance column is between 1000 and 5000 (inclusive). You can adjust the values as needed.

e) To create a trigger that activates when the balance is below 1000 in the Bank table, you can use the following SQL statement:

```sql
CREATE OR REPLACE TRIGGER balance_trigger
BEFORE INSERT OR UPDATE ON Bank
FOR EACH ROW
BEGIN
  IF :NEW.Balance < 1000 THEN
    -- Trigger actions here (e.g., raise an exception, log a message, etc.)
    -- Example action: Print a message to the console
```
```

```
    DBMS_OUTPUT.PUT_LINE('Balance is below 1000!');
  END IF;
END;
/
```

This trigger is executed before each insert or update operation on the Bank table. It checks if the new value of the Balance column is less than 1000, and if so, it performs the specified actions. In this example, it prints a message to the console using `DBMS_OUTPUT.PUT_LINE()`. You can customize the trigger actions as per your requirements.

---

9) For the table "Account", an appropriate primary key would be "Account_No". It uniquely identifies each account in the table.

a) To display the Cust_Name and Account_No of the customers from the branch "Branch = XXXXX", you can use the following SQL query:

```sql
SELECT Cust_Name, Account_No
FROM Account
WHERE Branch_Name = 'XXXXX';
```

Replace 'XXXXX' with the actual branch name.

b) To display the names and account types of all customers whose account balance is more than 10,000, you can use the following SQL query:

```sql
```

```
SELECT Cust_Name, Account_Type

FROM Account

WHERE Account_Balance > 10000;
```

This query selects the Cust_Name and Account_Type columns from the Account table where the Account_Balance is greater than 10,000.

c) To add the column "Cust_Date_of_Birth" to the "Account" table, you can use the following SQL statement:

```sql
ALTER TABLE Account

ADD Cust_Date_of_Birth DATE;
```

This adds a new column called "Cust_Date_of_Birth" of type DATE to the "Account" table.

d) To display the Account_No, Cust_Name, and Branch of all customers whose account balance is less than 1,000, you can use the following SQL query:

```sql
SELECT Account_No, Cust_Name, Branch_Name

FROM Account

WHERE Account_Balance < 1000;
```

This query selects the Account_No, Cust_Name, and Branch_Name columns from the Account table where the Account_Balance is less than 1,000.

e) Here's an example of a procedure that performs the above operations for the "Account" database:

```sql
CREATE OR REPLACE PROCEDURE AccountOperations
IS
BEGIN
  -- a) Display the Cust_Name and Account_No of the customers from a specific branch
  SELECT Cust_Name, Account_No
  INTO variable1, variable2 -- Declare appropriate variables
  FROM Account
  WHERE Branch_Name = 'XXXXX'; -- Replace 'XXXXX' with the actual branch name

  -- b) Display the names and account types of customers with account balance > 10,000
  SELECT Cust_Name, Account_Type
  INTO variable3, variable4 -- Declare appropriate variables
  FROM Account
  WHERE Account_Balance > 10000;

  -- c) Add the Cust_Date_of_Birth column to the Account table
  ALTER TABLE Account
  ADD Cust_Date_of_Birth DATE;
```

```
  -- d) Display Account_No, Cust_Name, and Branch of customers with account balance < 1,000

  SELECT Account_No, Cust_Name, Branch_Name

  INTO variable5, variable6, variable7 -- Declare appropriate variables

  FROM Account

  WHERE Account_Balance < 1000;

END;

/
```

Note: In the above code, you need to declare appropriate variables and handle any specific logic or output requirements for each operation.

You can execute the procedure using the following command:

```sql
BEGIN

  AccountOperations;

  DBMS_OUTPUT.PUT_LINE('Account operations executed successfully.');

END;

/
```

This will execute the procedure and display a message indicating the successful completion of the operations.

---

10) For the tables CUSTOMER and ORDER:

a) To list the names and addresses of customers who have ordered products costing more than 500, you can use the following SQL query:

```sql
SELECT c.Name, c.Address
FROM CUSTOMER c
JOIN ORDER o ON c.C_ID = o.C_ID
WHERE o.P_COST > 500;
```

This query joins the CUSTOMER and ORDER tables based on the C_ID column and selects the Name and Address columns from CUSTOMER where the P_COST in ORDER is greater than 500.

b) To list the names of products ordered with a cost of 1,000 or more, you can use the following SQL query:

```sql
SELECT P_Name
FROM ORDER
WHERE P_COST >= 1000;
```

This query selects the P_Name column from the ORDER table where the P_COST is greater than or equal to 1,000.

c) To list the product names ordered by customers from the city "Delhi", you can use the following SQL query:

```sql
SELECT o.P_Name
FROM CUSTOMER c
JOIN ORDER o ON c.C_ID = o.C_ID
WHERE c.City = 'Delhi';
```

This query joins the CUSTOMER and ORDER tables based on the C_ID column and selects the P_Name column from ORDER where the City in CUSTOMER is "Delhi".

d) To add a column "Email_id" in the CUSTOMER table, you can use the following SQL statement:

```sql
ALTER TABLE CUSTOMER
ADD Email_id VARCHAR(100);
```

This adds a new column called "Email_id" of type VARCHAR(100) to the CUSTOMER table.

e) To demonstrate a user-defined function for the above tables, you can create a function using the following example:

```sql
CREATE OR REPLACE FUNCTION CalculateTotalOrderCost(customer_id IN NUMBER)
RETURN NUMBER
IS
  total_cost NUMBER := 0;
BEGIN
  SELECT SUM(P_COST)
```

```
  INTO total_cost
  FROM "ORDER"
  WHERE C_ID = customer_id;
  RETURN total_cost;
END;
/
```

This function calculates the total order cost for a given customer ID. You can customize the logic based on your specific requirements. After creating the function, you can use it in SQL queries like:

```sql
SELECT C_ID, CalculateTotalOrderCost(C_ID) AS Total_Cost
FROM CUSTOMER;
```

This query retrieves the customer ID and the total order cost for each customer using the user-defined function.
