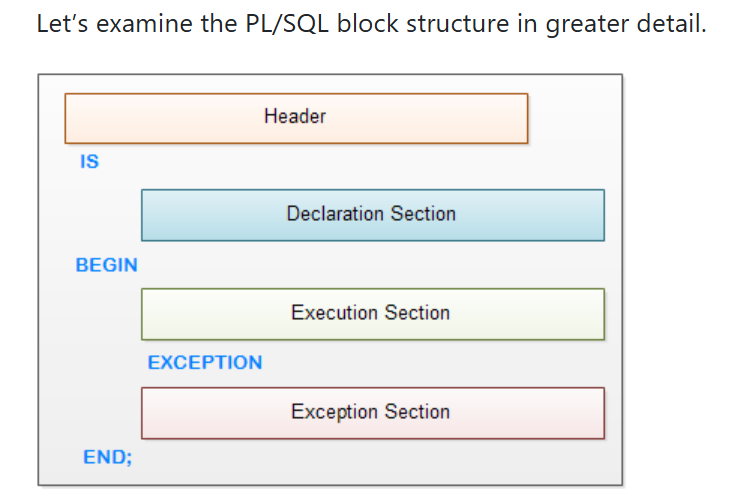
[Document.docx](https://1drv.ms/w/c/777eb826e035dfce/EbjBEL7Dj1FIhuVlg6lykeEBW4Wt9rvc30oR3sAsSTu7cA?e=AobGeR)

# 3. PL/SQL Architecture

- \*\*Anonymous Block\*\*: Code block not stored permanently in the database.  
- \*\*Named Blocks\*\*: Includes procedures, functions, triggers.

<https://www.plsqltutorial.com/plsql-block-structure/>

## PL/SQL Block Structure



The image shows the structure of a **PL/SQL block**, which consists of:

1. **Header** (optional — only in named blocks)
2. **Declaration Section** (DECLARE)
3. **Execution Section** (BEGIN ... END)
4. **Exception Section** (EXCEPTION)

Structure:

DECLARE  
 -- Variable declarations  
BEGIN  
 -- Executable statements  
EXCEPTION  
 -- Error handling logic  
END;

## What is the **Header** in PL/SQL?

### The **Header** is used only in **named blocks**, such as:

* **Procedures**
* **Functions**
* **Triggers**
* **Packages**

It is the part where you define the **name** of the block and, optionally, **parameters**.

### The Header is **not required** in anonymous blocks.

### Examples

### **Anonymous Block (No Header)**

DECLARE

v\_msg VARCHAR2(50);

BEGIN

v\_msg := 'Hello';

DBMS\_OUTPUT.PUT\_LINE(v\_msg);

END;

/

Note if we write directly it will give error DBMS\_OUTPUT.PUT\_LINE(v\_msg);

It should be in begin and end block

|  |  |  |
| --- | --- | --- |
| **Section** | Mandatory? | Description |
| **Declaration** | Optional | Declare variables/constants/cursors |
| **Execution (BEGIN)** | Yes | Main logic happens here |
| **Exception** | Optional | Handles errors if they occur during execution |

### **Named Block with Header (Procedure)**

| **Section** | **Mandatory?** | **Description** |
| --- | --- | --- |
| **Header** | Optional | Used for named blocks like procedures/functions |

CREATE OR REPLACE PROCEDURE greet\_user(p\_name VARCHAR2)

IS -- << Header ends here

v\_msg VARCHAR2(100);

BEGIN

v\_msg := 'Hello, ' || p\_name;

DBMS\_OUTPUT.PUT\_LINE(v\_msg);

END;

/

🔹 Here, greet\_user(p\_name VARCHAR2) is the **header** of the procedure.

## Summary Table

| **Section** | **Mandatory?** | **Description** |
| --- | --- | --- |
| **Header** | Optional | Used for named blocks like procedures/functions |
| **Declaration** | Optional | Declare variables/constants/cursors |
| **Execution (BEGIN)** | Yes | Main logic happens here |
| **Exception** | Optional | Handles errors if they occur during execution |

### Q: **What is** IS **in the PL/SQL procedure?**

**Is it mandatory? What does it do?**

### Answer:

In a **named PL/SQL block** (like a **procedure**, **function**, or **trigger**), the keyword IS (or AS) is used to **begin the definition** of the block **after the header**.

### **Syntax Pattern:**

CREATE [OR REPLACE] PROCEDURE procedure\_name (parameters)

IS

-- Declarations

BEGIN

-- Executable Statements

END;

or

CREATE [OR REPLACE] PROCEDURE procedure\_name (parameters)

AS

-- Declarations

BEGIN

-- Executable Statements

END;

### So, is IS mandatory?

**Yes — it is mandatory** in **named PL/SQL blocks**, such as:

* PROCEDURE
* FUNCTION
* TRIGGER

**But not used** in **anonymous blocks**.

### IS vs AS — Are they different?

They are **interchangeable** in most cases!

| **Keyword** | **Usage Context** | **Meaning** |
| --- | --- | --- |
| IS | More common in procedures | Start of the block body |
| AS | Also accepted (esp. in functions/triggers) | Same meaning |

Both IS and AS work — just **don’t skip them** in named blocks.

### Example with AS instead of IS:

CREATE OR REPLACE PROCEDURE greet\_user(p\_name VARCHAR2)

AS

v\_msg VARCHAR2(100);

BEGIN

v\_msg := 'Hello, ' || p\_name;

DBMS\_OUTPUT.PUT\_LINE(v\_msg);

END;

/

Same output, same functionality.

### Final Verdict:

* In **named blocks**, IS or AS is **mandatory**
* In **anonymous blocks**, you skip the header and **don't need IS**

Bottom of Form

The following example displays a message Hello PL/SQL on a screen using SQL\*Plus:

SET SERVEROUTPUT ON SIZE 1000000

BEGIN

DBMS\_OUTPUT.PUT\_LINE('Hello PL/SQL');

END;

/

## Why it works **without** SET SERVEROUTPUT ON in **Oracle Live SQL**

### In **Oracle Live SQL**:

* SET SERVEROUTPUT ON is **enabled by default** in the **web-based Oracle Live SQL environment**.
* So, **you don’t need to manually turn it on** like you do in SQL\*Plus or SQL Developer CLI.

That's why this runs **without issues**:

BEGIN

DBMS\_OUTPUT.PUT\_LINE('Hello PL/SQL');

END;

And it **shows output** like:

Hello PL/SQL

PL/SQL procedure successfully completed.

## But in other environments like:

### **SQL\*Plus**

### **SQL Developer (Worksheet)**

### **Toad / DBeaver / CLI**

You must explicitly run:

SET SERVEROUTPUT ON;

**Otherwise, you won’t see DBMS\_OUTPUT text** even though your PL/SQL block executes successfully.

* **SET SERVEROUTPUT ON is auto-enabled**, but still **show them the command** — because in real-world tools, **they will need it**.

# Variables and Data Types

## PL/SQL variables naming rules

Like other programming languages, a variable in PL/SQL must follow the naming rules as follows:

* The variable name must be less than 31 characters. Try to make it as meaningful as possible within 31 characters.
* The variable name must begin with an ASCII letter. It can be either lowercase or uppercase. Notice that PL/SQL is case-insensitive, which means v\_data and V\_DATA refer to the same variable.
* Followed by the first character are any number, underscore ( \_), and dollar sign ( $) characters. Once again, do not make your variables hard to read and difficult to understand.

## PL/SQL variables naming convention

It is highly recommended that you should follow the naming conventions listed in the following table to make the variables obvious in PL/SQL programs:

| **Prefix** | **Data Type** |
| --- | --- |
| v\_ | VARCHAR2 |
| n\_ | NUMBER |
| t\_ | TABLE |
| r\_ | ROW |
| d\_ | DATE |
| b\_ | BOOLEAN |

Each organization has its own development naming convention guidelines. Make sure that you comply with your organization’s naming convention guidelines.

For example, if you want to declare a variable that holds the first name of the employee with the VARCHAR2 data type, the variable name should be v\_first\_name.

## PL/SQL Variables Declaration

To declare a variable, you use a variable name followed by the data type and terminated by a semicolon ( ;). You can also explicitly add a length constraint to the data type within parentheses. The following illustrates some examples of declaring variables in a PL/SQL [anonymous block](https://plsqltutorial.com/plsql-block-structure/):

DECLARE

v\_first\_name varchar2(20);

v\_last\_name varchar2(20);

n\_employee\_id number;

d\_hire\_date date;

BEGIN

NULL;

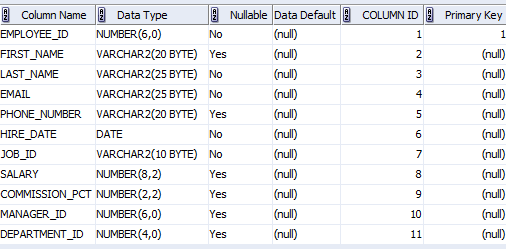
END;Code language: SQL (Structured Query Language) (sql)

## PL/SQL variable anchors

In PL/SQL program, one of the most common tasks is to select values from columns in a table into a set of variables. In case the data types of columns of the table changes, you have to change the PL/SQL program to make the types of the variables compatible with the new changes.

PL/SQL provides you with a very useful feature called **variable anchors**. It refers to the use of the  %TYPE  keyword to declare a variable with the data type is associated with a column’s data type of a particular column in a table.

Let’s take a look at the employeestable in HRsample database provided by Oracle:

[](https://plsqltutorial.com/wp-content/uploads/2011/05/Employees_Table.gif)Employees Table

DECLARE

v\_first\_name EMPLOYEES.FIRST\_NAME%TYPE;

v\_last\_name EMPLOYEES.LAST\_NAME%TYPE;

n\_employee\_id EMPLOYEES.EMPLOYEE\_ID%TYPE;

d\_hire\_date EMPLOYEES.HIRE\_DATE%TYPE;

BEGIN

NULL;

END;

/Code language: SQL (Structured Query Language) (sql)

The v\_first\_name variable has a data type that is the same as the data type of the first\_namecolumn in the  emloyees  table. In case the data type of the first\_namecolumn changes, the type of the v\_first\_namevariable automatically inherits the new data type of the column.

## PL/SQL variable assignment

In PL/SQL, to assign a value or a variable to another, you use the assignment operator ( := ) which is a colon( :) followed by the equal sign( = ).

Please see the code listing below to get a better understanding:

DECLARE

v\_first\_name EMPLOYEES.FIRST\_NAME%TYPE;

v\_last\_name EMPLOYEES.LAST\_NAME%TYPE;

n\_employee\_id EMPLOYEES.EMPLOYEE\_ID%TYPE;

d\_hire\_date EMPLOYEES.HIRE\_DATE%TYPE;

BEGIN

v\_first\_name := 'Mary';

v\_last\_name := 'Jane';

d\_hire\_date := to\_date('19700101','YYYYMMDD');

END;

/Code language: SQL (Structured Query Language) (sql)

In the example above, we assigned Mary to v\_first\_namevariable, Janeto v\_last\_namevariable, and result of the to\_datefunction to d\_hire\_datevariable.

You can use INTOof the SELECT statement to assign a value to a variable. The INTOclause moves the values from the SELECTquery’s column list into corresponding PL/SQL variables.

SET SERVEROUTPUT ON SIZE 1000000;

DECLARE

v\_first\_name EMPLOYEES.FIRST\_NAME%TYPE;

v\_last\_name EMPLOYEES.LAST\_NAME%TYPE;

n\_employee\_id EMPLOYEES.EMPLOYEE\_ID%TYPE;

d\_hire\_date EMPLOYEES.HIRE\_DATE%TYPE;

BEGIN

SELECT employee\_id,

first\_name,

last\_name,

hire\_date

INTO n\_employee\_id,

v\_first\_name,

v\_last\_name,

d\_hire\_date

FROM employees

WHERE employee\_id = 200;

DBMS\_OUTPUT.PUT\_LINE(v\_first\_name);

DBMS\_OUTPUT.PUT\_LINE(v\_last\_name);

DBMS\_OUTPUT.PUT\_LINE(d\_hire\_date);

END;

/Code language: SQL (Structured Query Language) (sql)

## Initializing variables

When you declare a variable, its value is uninitialized and hence is NULL. You can initialize variable a value in declaration section by using variable assignment.

See the following example:

DECLARE

n\_employee\_id EMPLOYEES.EMPLOYEE\_ID%TYPE :=200;

d\_hire\_date EMPLOYEES.HIRE\_DATE%TYPE:=to\_date('19700101','YYYYMMDD');

BEGIN

NULL;

END;

/Code language: SQL (Structured Query Language) (sql)

In PL/SQL, NULLmeans an unknown value so it has some special characteristics as follows:

* NULLis not equal to anything, even itself NULL.
* NULLis not greater than or less than anything else, even NULL.
* You cannot use logical operator equal ( =) or ( <>) with NULL. You must use the SQL IS NULL or  IS NOT NULL  to test the NULL values.

# PL/SQL Collections

Collections are single-dimension data structures (like arrays). Types:  
- \*\*Associative Arrays\*\* (index-by tables): It is in-memory Unbounded sets of key-value pairs.

- \*\*VARRAYs\*\*: Fixed-size arrays.  
- \*\*Nested Tables\*\*: Variable-size arrays that can be stored in DB tables.

in PL/SQL, the keyword TABLE + INDEX BY = **Associative Array**

This is the **official syntax for associative arrays** in PL/SQL:

TYPE <type\_name> IS TABLE OF <data\_type> INDEX BY <key\_type>;

It’s called "table" for historical reasons (like index-by tables) — but it is **not a SQL table**.

Important: Associative Arrays are the **only collection type** in PL/SQL that use INDEX BY.

## What is TYPE in PL/SQL?

### It is a way to define **user-defined (custom) data types** — particularly:

* **Collections** (like arrays)
* **Records** (like structs or composite types)
* **Object types** (in advanced use)

## 1. **"Associative Array is in-memory" — but where exactly?**

### Where is it stored?

Associative arrays in PL/SQL are stored in **RAM (Random Access Memory)** — specifically, in **the server-side memory allocated to your PL/SQL session**.

### So:

* They **do not persist** in the database.
* They exist **only during the session** or block execution.
* Once your procedure/block ends, or session disconnects → the array is gone.

**Not stored in ROM** (Read-Only Memory), and not written to disk unless you manually save the values into a table.

## 2. What does **unbounded in size** mean?

### Unbounded means:

There is **no fixed limit** or maximum number of elements for the collection.

### For Example:

DECLARE

TYPE student\_list IS TABLE OF VARCHAR2(100) INDEX BY PLS\_INTEGER;

students student\_list;

BEGIN

FOR i IN 1..100000 LOOP

students(i) := 'Student ' || i;

END LOOP;

END;

* You didn’t declare a maximum size (VARRAY(1000)), but it can grow as long as memory allows.
* So, it is **only limited by available server memory** (RAM).

Unlike VARRAY, which is fixed (e.g., VARRAY(5)), **associative arrays grow as needed**.

## 3. What is PLS\_INTEGER?

### PLS\_INTEGER is a **PL/SQL-only datatype** (not a SQL datatype).

| **Feature** | **PLS\_INTEGER** |
| --- | --- |
| Type | Integer |
| Range | –2,147,483,648 to +2,147,483,647 |
| Speed | Fastest integer type in PL/SQL |
| Use | For indexing or fast math operations |
| Storage | Only in **PL/SQL memory**, not in tables |

It is faster than NUMBER in PL/SQL because it uses native machine arithmetic.

Note \* You can also index by string:

TYPE student\_grades IS TABLE OF VARCHAR2(10) INDEX BY VARCHAR2(20);

Then:

grades('EMP001') := 'A';

grades('EMP002') := 'B';

## 4. Is indexing done automatically in associative arrays?

### Not exactly "automatically" — **you control the keys** (indexes).

In associative arrays:

* **You decide the index values.**
* If you use INDEX BY PLS\_INTEGER, you insert like: my\_array(1) := 'value';
* If you use INDEX BY VARCHAR2(100), you insert like: my\_array('emp001') := 'John';

### Example:

DECLARE

TYPE grade\_table IS TABLE OF VARCHAR2(10) INDEX BY PLS\_INTEGER;

grades grade\_table;

BEGIN

grades(1) := 'A';

grades(100) := 'B'; -- You decide the index!

grades(500) := 'C';

DBMS\_OUTPUT.PUT\_LINE(grades(100)); -- Output: B

END;

No default indexing like a normal array starting from 0 or 1. You manually assign keys.

## Summary

| **Concept** | **Explanation** |
| --- | --- |
| **In-memory?** | Yes, stored in **RAM (PGA memory)**, only during session |
| **Persistent?** | No, associative arrays disappear after block/session ends |
| **Unbounded?** | Yes, grows as needed — limited only by server memory |
| **PLS\_INTEGER?** | Fastest PL/SQL integer type; used for indexing |
| **Indexing?** | You provide keys (integer or string), nothing is auto-indexed |

## Summary of Keywords

| **Keyword** | **Role** |
| --- | --- |
| TYPE | Creates a **new PL/SQL type** (in this case, a collection) |
| IS TABLE OF | Declares a **collection of values** |
| VARCHAR2(10) | Each element will be a VARCHAR2 value (max 10 characters) |
| INDEX BY | Means you're defining an **Associative Array** |
| PLS\_INTEGER | The datatype for the **index/key** (faster than NUMBER) |

## Associative Arrays – Example Use Case

## Example 1: Defining a **custom collection** type (Associative Array)

sql

CopyEdit

DECLARE

TYPE grade\_table IS TABLE OF VARCHAR2(10) INDEX BY PLS\_INTEGER;

grades grade\_table;

BEGIN

grades(1) := 'A';

grades(2) := 'B';

DBMS\_OUTPUT.PUT\_LINE(grades(2));

END;

* TYPE grade\_table → You are creating a **new custom data type**
* grades grade\_table; → You’re declaring a variable **of that custom type**

### Why is it still **unbounded**?

Even though you say VARCHAR2(10), you're **defining the data type of each element**, not the number of elements.

So:

* 'VARCHAR2(10)' means: each value in the array can be **up to 10 characters long**
* **It does NOT mean** the array is limited to 10 elements

Associative arrays are unbounded because:

* You can keep inserting any number of values
* You can use any valid key (1, 100, 500, 999999, etc.)
* There’s **no fixed limit** or MAXSIZE like VARRAY

## What does VARCHAR2(10) mean here?

* It means **each value in the associative array can hold up to 10 characters.**
* So you’re telling Oracle:

“Every element of my collection is a string, with max length = 10 characters.”

## Why would we restrict size like this?

To:

1. **Optimize memory**: If you're storing codes, grades, short strings (like 'Pass', 'Fail', 'Grade A+'), you don’t need 100 or 1000 characters.
2. **Prevent overflows**: You want to avoid storing unintended long text.

### Real-World Example:

Imagine you're storing **student grades**, and the values can be like:

* "A+"
* "B-"
* "Distinction" → that's 11 characters (would raise an error if you insert into a VARCHAR2(10) field)

## Demonstration Code:

DECLARE

TYPE grade\_table IS TABLE OF VARCHAR2(10) INDEX BY PLS\_INTEGER;

grades grade\_table;

BEGIN

grades(1) := 'A+'; -- OK (2 characters)

grades(2) := 'Distinction'; -- Error: string length exceeds 10

DBMS\_OUTPUT.PUT\_LINE(grades(1));

DBMS\_OUTPUT.PUT\_LINE(grades(2)); -- Will not run due to the above error

END;

### Error you’ll get:

ORA-06502: PL/SQL: numeric or value error: character string buffer too small

## Can we store both **int** and **string** values in the same associative array?

**No.** You can only store **one data type per collection** in PL/SQL.

## Example (Correct Usage — Single Type):

DECLARE

TYPE grades\_type IS TABLE OF VARCHAR2(10) INDEX BY PLS\_INTEGER;

grades grades\_type;

BEGIN

grades(1) := 'A+';

grades(2) := 'B';

-- grades(3) := 100; -- Error: assigning a number to VARCHAR2

DBMS\_OUTPUT.PUT\_LINE(grades(1));

END;

grades(3) := 100;

will raise **ORA-06550** because you're assigning a NUMBER to a VARCHAR2 array.

## Syntax: Associative Array for Integer Values

TYPE array\_name IS TABLE OF <numeric\_datatype> INDEX BY PLS\_INTEGER;

* NUMBER → General numeric type (can be integer or decimal)
* PLS\_INTEGER → Optimized integer type for PL/SQL (faster in memory)

### Example 1: Storing Marks (Using NUMBER)

DECLARE

TYPE marks\_table IS TABLE OF NUMBER INDEX BY PLS\_INTEGER;

student\_marks marks\_table;

BEGIN

student\_marks(101) := 85;

student\_marks(102) := 92;

student\_marks(103) := 77;

DBMS\_OUTPUT.PUT\_LINE('Student 102 Score: ' || student\_marks(102));

END;

Output:

Student 102 Score: 92

### Example 2: Using PLS\_INTEGER for Better Performance

DECLARE

TYPE int\_array IS TABLE OF PLS\_INTEGER INDEX BY PLS\_INTEGER;

values\_array int\_array;

BEGIN

values\_array(1) := 10;

values\_array(2) := 20;

values\_array(5) := 100;

DBMS\_OUTPUT.PUT\_LINE('Value at index 5: ' || values\_array(5));

END;

## PLS\_INTEGER vs. NUMBER

| **Feature** | **PLS\_INTEGER** | **NUMBER** |
| --- | --- | --- |
| Type | Whole numbers only | Supports decimals too |
| Performance | Fast (native) | Slower in PL/SQL |
| Range | ±2 billion | Much wider |
| Use For | Loop counters, indexes | Prices, salaries, etc. |

## Summary

To store integers in associative arrays:

| **Use Case** | **Data Type to Use** | **Example** |
| --- | --- | --- |
| Any numeric (int/float) | NUMBER | TABLE OF NUMBER INDEX BY ... |
| Fast integer-only ops | PLS\_INTEGER | TABLE OF PLS\_INTEGER INDEX BY ... |

### Option 2: **Use two separate arrays**

DECLARE

TYPE name\_table IS TABLE OF VARCHAR2(100) INDEX BY PLS\_INTEGER;

TYPE score\_table IS TABLE OF NUMBER INDEX BY PLS\_INTEGER;

names name\_table;

scores score\_table;

BEGIN

names(1) := 'Alice';

scores(1) := 85;

names(2) := 'Bob';

scores(2) := 90;

DBMS\_OUTPUT.PUT\_LINE(names(1) || ': ' || scores(1));

END;

Exercise for learners

Create a associative array to store learners outstanding performance as excellent ,average and poor

For eg:

DECLARE

TYPE performance\_table IS TABLE OF VARCHAR2(10) INDEX BY PLS\_INTEGER; -- declaring or creating a custom defined type of an associative array

performance performance\_table; --  This declares a **variable** grades of the new collection type you

defined above.

 performance is now your associative array.

BEGIN

performance(1) := 'Excellent ';

performance(100) := ‘Average';

performance(500) := 'Poor';

DBMS\_OUTPUT.PUT\_LINE(performance(100));

END;

Now see a record type

## Example 2: Defining a **custom record** type

DECLARE

TYPE emp\_record\_type IS RECORD (

emp\_id NUMBER,

emp\_name VARCHAR2(50)

);

v\_emp emp\_record\_type;

BEGIN

v\_emp.emp\_id := 101;

v\_emp.emp\_name := 'John';

DBMS\_OUTPUT.PUT\_LINE(v\_emp.emp\_id || ' - ' || v\_emp.emp\_name);

END;

Here, emp\_record\_type is a **custom composite datatype**, and v\_emp is a variable of that record type.

## So Yes — TYPE in PL/SQL:

| **Concept** | **Explanation** |
| --- | --- |
| Custom Data Type | Yes, created using TYPE keyword |
| Reusable? | Yes, within the block/package where it’s defined |
| Used For | Collections, records, object types |
| Similar To | Struct in C/C++, Class without methods, Table definition |

## **In Associative Arrays (INDEX BY)** — Do indexes start from 0 or follow any order?

### **No, Associative Arrays in PL/SQL do NOT require sequential indexing.**

## Associative Arrays are **sparse** and **flexible**:

* You can use **any valid integer as a key** — positive or negative.
* The array **does not require keys to start at 0 or 1**.
* You can have keys like: grades(-5), grades(100), grades(9999).

## Real Examples:

DECLARE

TYPE grade\_table IS TABLE OF VARCHAR2(10) INDEX BY PLS\_INTEGER;

grades grade\_table;

BEGIN

grades(100) := 'A';

grades(5000) := 'B';

grades(-3) := 'F';

DBMS\_OUTPUT.PUT\_LINE(grades(100)); -- A

DBMS\_OUTPUT.PUT\_LINE(grades(-3)); -- F

END;

You inserted values at keys 100, 5000, and –3 — and it worked fine.

## So Why is it Called "Index"?

In PL/SQL, the **index** just means the **key** used to refer to a value in the array.

But unlike regular arrays in other languages:

* It’s **not automatically assigned**
* It’s **not required to be sequential**

## Can I start from 0?

Yes. You can use 0 if you like:

grades(0) := 'Pass';

PL/SQL does **not assume a default starting index** — you decide what to use.

## Summary Table

| **Feature** | **Associative Array in PL/SQL** |
| --- | --- |
| Starts from index 0? | No, you decide the index |
| Must be sequential? | No, can be 1, 100, –10, 5000 |
| Must be unique keys? | Yes, each index must be unique |
| Can use negative keys? | Yes |
|  |  |

Use Case: Store last employee entered for each department. We will see when we will learn for and loop

Example:

DECLARE  
 TYPE emp\_table\_type IS TABLE OF VARCHAR2(100) INDEX BY PLS\_INTEGER;  
 v\_emps emp\_table\_type;  
BEGIN  
 FOR r IN (SELECT dept\_id, emp\_name FROM employees) LOOP  
 v\_emps(r.dept\_id) := r.emp\_name;  
 END LOOP;  
 DBMS\_OUTPUT.PUT\_LINE('Dept 10 Employee: ' || v\_emps(10));  
END;

Create an **object type** representing a product with:

* Product ID
* Product Name
* Price

Then create a **collection (nested table)** of these product objects, insert values, and display them using a PL/SQL block.

## Step-by-Step Example: Nested Table of Object Type

### Step 1: Create the Object Type

CREATE OR REPLACE TYPE product\_obj AS OBJECT (

product\_id NUMBER,

product\_name VARCHAR2(50),

price NUMBER

);

### Step 2: Create a Collection Type (Nested Table of the Object)

CREATE OR REPLACE TYPE product\_table AS TABLE OF product\_obj;

**Step 3: Use the Object Array in PL/SQL**

DECLARE

-- Declare a nested table of product\_obj

v\_products product\_table := product\_table(); -- initialize

BEGIN

-- Add 3 products

v\_products.EXTEND(3);

v\_products(1) := product\_obj(101, 'Sugar', 45.50);

v\_products(2) := product\_obj(102, 'Milk', 30.00);

v\_products(3) := product\_obj(103, 'Rice', 70.00);

-- Print the product list

FOR i IN 1 .. v\_products.COUNT LOOP

DBMS\_OUTPUT.PUT\_LINE(

'Product ID: ' || v\_products(i).product\_id || ', ' ||

'Name: ' || v\_products(i).product\_name || ', ' ||

'Price: ₹' || v\_products(i).price

);

END LOOP;

END;

## Output:

Product ID: 101, Name: Sugar, Price: ₹45.5

Product ID: 102, Name: Milk, Price: ₹30

Product ID: 103, Name: Rice, Price: ₹70

## 🧠 Summary

| **Component** | **Purpose** |
| --- | --- |
| OBJECT type | Defines the structure of a record |
| Nested Table | Stores multiple object-type values |
| EXTEND | Grows the collection |
| product\_obj(...) | Initializes an object instance |

### ✅ Real-World Use Case

* Used in Oracle forms and APIs
* Ideal for passing structured data (like cart items, orders, etc.)
* Used in stored procedures, especially in PL/SQL web apps or ERP systems

Let me know if you'd like:

* A version that inserts this object array into a table
* Or converts it into a **VARRAY** instead of nested table

Happy to help!

##### You said:

A version that inserts this object array into a table -- yes how to store in a table give the complete table structure

###### ChatGPT said:

Great! Let's now **store a collection of object types (object array)** into a **relational table**. This is a very practical scenario — often used in real-world systems like order processing, invoices, etc.

## 🎯 Objective:

You’ll:

1. Create an **object type** product\_obj
2. Create a **nested table type** product\_table (a collection of objects)
3. Create a **master table** with a column of that object array type
4. **Insert multiple products** into that column
5. Use a **SELECT with TABLE()** to view individual records

## ✅ Step-by-Step Implementation (Oracle Live SQL Compatible)

### 🔹 Step 1: Create the Object Type

sql

CopyEdit

CREATE OR REPLACE TYPE product\_obj AS OBJECT (

product\_id NUMBER,

product\_name VARCHAR2(50),

price NUMBER

);

### 🔹 Step 2: Create the Collection Type (Nested Table of Object)

sql

CopyEdit

CREATE OR REPLACE TYPE product\_table AS TABLE OF product\_obj;

### 🔹 Step 3: Create a Table That Uses the Collection

sql

CopyEdit

CREATE TABLE order\_master (

order\_id NUMBER PRIMARY KEY,

customer\_name VARCHAR2(50),

products product\_table

)

NESTED TABLE products STORE AS product\_nested\_tab;

STORE AS defines the storage for the nested table's inner structure.

### 🔹 Step 4: Insert Data into the Table

sql

CopyEdit

BEGIN

INSERT INTO order\_master VALUES (

1,

'Ravi',

product\_table(

product\_obj(101, 'Sugar', 45.50),

product\_obj(102, 'Milk', 30.00),

product\_obj(103, 'Rice', 70.00)

)

);

INSERT INTO order\_master VALUES (

2,

'Neha',

product\_table(

product\_obj(104, 'Bread', 25.00),

product\_obj(105, 'Jam', 50.00)

)

);

END;

### 🔹 Step 5: Query the Nested Table (Unnesting)

To view product items from a specific order:

sql

CopyEdit

SELECT o.order\_id, o.customer\_name, p.product\_id, p.product\_name, p.price

FROM order\_master o,

TABLE(o.products) p;

## 🧾 Expected Output:

| **ORDER\_ID** | **CUSTOMER\_NAME** | **PRODUCT\_ID** | **PRODUCT\_NAME** | **PRICE** |
| --- | --- | --- | --- | --- |
| 1 | Ravi | 101 | Sugar | 45.50 |
| 1 | Ravi | 102 | Milk | 30.00 |
| 1 | Ravi | 103 | Rice | 70.00 |
| 2 | Neha | 104 | Bread | 25.00 |
| 2 | Neha | 105 | Jam | 50.00 |

## ✅ Summary

| **Step** | **Code/Action** |
| --- | --- |
| Create Object Type | product\_obj |
| Create Nested Table Type | product\_table |
| Table with Nested Column | order\_master (products product\_table) |
| Insert Data | Using product\_table(...) constructor |
| Query Nested Table | Use TABLE() to unnest collection |

Would you like this exported as a .sql script file for Oracle Live SQL?

Or would you like to expand this example into:

* An **order total calculation function**
* Or **update/append products** to existing orders?