PARADIGM **OF PROGRAMMING**

1. **Procedural Programming Paradigm**

**Definition**:  
Procedural programming is a subset of the **imperative paradigm** that structures programs as a series of *procedures* (also called routines, subroutines, or functions). It's one of the oldest and most widely used programming paradigms.

**🔧 Core Concepts**

* **Procedures/Functions** – Blocks of reusable code that perform specific tasks.
* **Sequence** – Code is executed in a top-down order.
* **Selection** – Conditional logic (e.g., if, else) to choose paths.
* **Iteration** – Loops (for, while) to repeat actions.
* **Variables and State** – Data is stored in variables and modified over time.

**🧠 How It Works**

You write a main procedure that calls other procedures in a specific order to perform a task. Data is usually passed around via parameters or global variables.

**✅ Advantages**

* Simple and intuitive for small to medium-sized programs.
* Encourages modularity through function reuse.
* Easier to debug due to a clear control flow.

**❌ Disadvantages**

* Poor data abstraction; everything is exposed unless carefully managed.
* Difficult to scale or maintain for large applications.
* Global state and side effects can lead to bugs.
* .

### 🧪 ****Example (in C)****

c

CopyEdit

#include <stdio.h>

// Procedure to add two numbers

int add(int a, int b) {

return a + b;

}

int main() {

int x = 5, y = 3;

int result = add(x, y);

printf("Sum is: %d\n", result);

return 0;

}

### 📌 Common Procedural Languages:

* **C**
* **Pascal**
* **Fortran**
* **BASIC**

Procedural programming is still very relevant, especially in system programming and embedded systems.

Would you like a comparison between procedural and object-oriented paradigms?

Top of Form



Bottom of Form

### 2. ****Declarative Programming Paradigm****

* **Core Idea**: Focuses on what the program should accomplish without specifying how.
* **How It Works**: You write expressions that describe logic or desired results; the system figures out how to compute them.
* **Example Languages**: SQL, HTML, Prolog.
* **Analogy**: Like telling a chef what dish you want, not how to cook it.
* **Pros**: Often more concise, easier to reason about.
* **Cons**: Less control over performance or execution details.

### 3. ****Object-Oriented Programming Paradigm (OOP)****

* **Core Idea**: Models programs as collections of objects, which combine data and behavior.
* **How It Works**: Uses classes and objects, with key concepts like encapsulation, inheritance, and polymorphism.
* **Example Languages**: Java, C++, Python (supports OOP).
* **Analogy**: Like modeling a system with real-world entities (e.g., a "Car" class with speed and move functions).
* **Pros**: Good for managing large, complex software; encourages reuse.
* **Cons**: Can lead to over-engineering; deep inheritance chains can be problematic.

### 4. ****Functional Programming Paradigm****

* **Core Idea**: Emphasizes pure functions, immutability, and avoiding side effects.
* **How It Works**: Treats computation as evaluation of mathematical functions; often uses recursion over loops.
* **Example Languages**: Haskell, Lisp, Elixir; also supported in Python, JavaScript.
* **Analogy**: Like solving equations in math.
* **Pros**: Easier to debug and test; naturally supports parallelism.
* **Cons**: Can be harder to understand for those used to imperative thinking; performance can vary.

### 5. ****Logic Programming Paradigm****

* **Core Idea**: Programs are a set of logical statements, and computation is done via inference.
* **How It Works**: You define rules and facts, and the system answers queries based on them.
* **Example Languages**: Prolog.
* **Analogy**: Like setting up a puzzle and letting the computer solve it.
* **Pros**: Good for AI and problem-solving domains.
* **Cons**: Less intuitive for general-purpose programming