ASSIGNMENT-1

1. What is the fundamental difference between procedural and object-oriented programming paradigms? Provide a brief example to illustrate

| **Feature** | **Procedural Programming** | **Object-Oriented Programming (OOP)** |
| --- | --- | --- |
| **Structure** | Follows a **top-down** approach, where the program is divided into functions | Follows a **bottom-up** approach, organizing code into objects |
| **Data Handling** | Functions operate on **global data**, making it vulnerable to unexpected modifications | Encapsulates **data within objects**, restricting unauthorized access |
| **Modularity** | Relies on **functions**, which can be reused but often require modifications to fit different contexts | Uses **classes and objects**, enhancing code reusability and reducing redundancy |
| **Security** | Less secure because variables are globally accessible | More secure due to encapsulation and access modifiers (public, private, protected) |
| **Scalability** | Not highly scalable, making large applications harder to manage | Highly scalable, ideal for complex and evolving software systems |
| **Abstraction** | Minimal abstraction; implementation details are exposed | Strong abstraction; internal logic can be hidden from users |
| **Inheritance** | No concept of inheritance; functions must be redefined | Supports inheritance, allowing reuse of existing functionalities |
| **Example Languages** | C, Pascal, FORTRAN | C++, Java, Python (OOP), C# |

2. Define Object-Oriented Programming (OOP). What are its core characteristics?

Object-Oriented Programming (OOP) is a programming paradigm that organizes software design around **objects** rather than functions or procedures. Objects are instances of **classes** that encapsulate data and behavior, promoting modularity, reusability, and efficiency in code development.

Core Characteristics of OOP

OOP is built upon four fundamental principles:

1. Encapsulation – Bundling data (variables) and methods (functions) into a single unit (object) while restricting direct access to certain details. *Example:* In a banking application, account details are encapsulated in an Account class, preventing unauthorized modifications.
2. Abstraction – Hiding complex implementation details and exposing only the necessary functionalities. *Example:* When using a car, you only interact with its controls (steering, pedals) without knowing the internal mechanics.
3. Inheritance – Allowing a class (child) to derive properties and behavior from another class (parent), reducing redundancy. *Example:* A Car class can inherit general attributes from a Vehicle class, like wheels and engine.
4. Polymorphism – Enabling objects to take multiple forms by allowing a common interface to be used for different types. *Example:* A draw() function could be used for different shapes (circle, square, triangle) without needing separate implementations.

**3.Explain the concept of "abstraction" within the context of OOP. Why is it important**

Abstraction is the process of **hiding unnecessary details** and exposing only the essential features of an object. In OOP, it allows developers to focus on what an object **does** rather than how it **works** internally.

### **Why is Abstraction important?**

1. Simplifies complexity– By hiding implementation details, abstraction makes software easier to understand and use.
2. **Enhances Maintainability** – Reduces dependencies on internal logic, allowing modifications without affecting other parts of the program.
3. **Improves Security** – Limits direct access to internal workings, preventing accidental or malicious alterations.
4. **Supports Code Reusability** – Helps developers use objects without needing to worry about their internal mechanics.

### 4.****Benefits of Object-Oriented Programming (OOP) Over Procedural Programming****

OOP offers several advantages over procedural programming, making it more suitable for large-scale and complex software development. Here are the key benefits:

1. **Encapsulation Enhances Security**
   * OOP encapsulates data within objects, restricting direct access and reducing the chances of unintended modifications.
   * Procedural programming often relies on global variables, making data more vulnerable.
2. **Modularity & Code Reusability**
   * In OOP, classes and objects can be reused across different projects, reducing redundancy.
   * Procedural programming requires copying and modifying functions for reuse.
3. **Scalability & Maintainability**
   * Large applications are easier to manage with OOP due to well-defined class structures.
   * Procedural code can become complex and difficult to update or scale.
4. **Abstraction Simplifies Complex Systems**
   * OOP allows programmers to expose only necessary details while hiding internal complexities.
   * Procedural programming requires users to understand entire function implementations.
5. **Polymorphism Improves Flexibility**
   * OOP enables a single interface to handle multiple implementations.
   * Procedural programming requires separate functions for each type of operation.
6. **Inheritance Reduces Redundancy**
   * OOP allows child classes to inherit properties from parent classes, saving development time.
   * Procedural programming lacks inheritance, requiring code duplication.
7. **Better Collaboration & Team Development**
   * OOP's modular design allows different teams to work on separate classes without conflicts.
   * Procedural programming’s sequential execution makes collaboration more challenging.

**5. Give a real-world example of a problem that is well-suited to be solved using an OOP approach. Explain why.**

### **Example: E-Commerce System**

An **online shopping platform** (like Amazon or Flipkart) is an excellent case where OOP helps structure a complex system efficiently.

### **Why OOP Works Well for E-Commerce Applications?**

1. **Encapsulation for Secure Transactions** – Customer payment details are kept private within objects, ensuring security.
2. **Modular & Scalable Design** – Different features (cart, orders, reviews) can be developed independently using classes.
3. **Inheritance for Efficient Management** – Electronics, Clothing, and Books can inherit common attributes from a Product class.
4. **Polymorphism for Flexibility** – Different payment methods (credit card, PayPal, UPI) can implement a common processPayment() method with their unique logic.
5. **Abstraction for Simplicity** – Users interact with high-level functions like addToCart() and placeOrder() without knowing the backend complexities.

6. **Define the four key principles of OOP: Encapsulation, Inheritance, Polymorphism, and Abstraction**

### **Four Key Principles of Object-Oriented Programming (OOP)**

OOP is built on four fundamental principles that promote modularity, security, and reusability in software development. Here’s what they mean:

1. **Encapsulation** – The practice of bundling data and methods within a single unit (class) while restricting direct access to certain details.
   * It prevents external interference and ensures controlled interaction with an object.
   * **Example: I**n a banking system, a BankAccount class can encapsulate account details, allowing controlled access through defined methods like getBalance() instead of direct variable manipulation.
2. **Inheritance** – A mechanism where a class (child) can inherit properties and behaviors from another class (parent), promoting code reuse.
   * Reduces redundancy and enables hierarchical relationships.
   * **Example:** A Car class can inherit common attributes from a Vehicle class, so every car automatically gets properties like wheels and engine without rewriting them.
3. **Polymorphism –** The ability of objects to take multiple forms by allowing a common interface to be used for different types.
   * Enhances flexibility and simplifies code maintenance.
   * **Example:** A draw() function can be applied to different shapes (circle, square, triangle), allowing each shape to define its own version of the function while using the same interface.
4. **Abstraction –** The concept of hiding complex implementation details while exposing only essential functionalities.
   * Simplifies user interaction and protects critical logic from direct exposure.
   * **Example:** A Car class can provide a startEngine() method without revealing the internal mechanical process—users simply press a button to start the engine without needing to understand the mechanics.

**7.Explain how encapsulation helps to protect data and create modular code. Give an example using a class and its members**.

Encapsulation is a fundamental principle of OOP **that protects data** and ensures **modularity** by restricting direct access to an object's internal state and requiring controlled interaction through methods.

### **How Encapsulation Helps Protect Data?**

1. **Restricts Unauthorized Access** – Private data members cannot be accessed directly, preventing accidental modifications.
2. **Maintains Data Integrity** – Changes to data happen through controlled methods, reducing errors.
3. **Enhances Security** – Sensitive information remains hidden from external interference.

### **How Encapsulation Supports Modular Code?**

1. **Encapsulated Objects Operate Independently** – Each class handles its own data, simplifying program structure.
2. **Improves Code Reusability** – Encapsulated classes can be reused without exposing internal details.
3. **Facilitates Maintenance & Debugging** – Changes within a class don’t affect other parts of the code.

#include <iostream>

using namespace std;

class Employee {

private:

int employeeID; // Private data member

double salary;

public:

Employee(int id, double sal) {

employeeID = id;

salary = sal;

}

void setSalary(double sal) {

if (sal > 0) {

salary = sal; // Controlled modification

} else {

cout << "Invalid salary amount!" << endl;

}

}

double getSalary() { // Controlled access

return salary;

}

};

int main() {

Employee emp(101, 50000);

// Accessing private data through public methods

cout << "Current Salary: ₹" << emp.getSalary() << endl;

emp.setSalary(60000); // Updating salary safely

cout << "Updated Salary: ₹" << emp.getSalary() << endl;

// Trying to access private variable directly (will cause error)

// cout << emp.salary;

return 0;

}

**8.** **What is inheritance? How does it promote code reuse and maintainability? Provide a simple example using classes.**

Inheritance is an OOP principle that allows a child cl**ass** to acquire properties and behaviors from a parent class. It promotes code reuse by eliminating redundancy and enhances maintainability by enabling modifications to be made in a single place, which automatically applies to all inherited classes.

How Inheritance Promotes Code Reuse & Maintainability?

1. Avoids Code Duplication – Instead of rewriting the same attributes and methods across multiple classes, inheritance lets new classes reuse existing logic from a parent class.
2. Facilitates Updates & Scalability – Changes made in a parent class automatically apply to all child classes, reducing the need for repetitive modifications.
3. Creates Hierarchical Structures – Allows logical grouping of objects, improving organization and readability.

Example of Inheritance in C++

cpp

#include <iostream>

using namespace std;

// Base class (Parent)

class Vehicle {

public:

int wheels;

void displayInfo() {

cout << "This is a vehicle with " << wheels << " wheels." << endl;

}

};

// Derived class (Child)

class Car : public Vehicle { // Car inherits Vehicle

public:

string brand;

void showBrand() {

cout << "This car is a " << brand << "." << endl;

}

};

int main() {

Car myCar;

myCar.wheels = 4; // Inherited attribute

myCar.brand = "Toyota"; // Child class attribute

myCar.displayInfo(); // Inherited method

myCar.showBrand(); // Child class method

return 0;

}

Key Takeaways from This Example

✅ Car inherits attributes (wheels) and methods (displayInfo()) from Vehicle. ✅ No need to redefine displayInfo() in Car, showcasing code reuse. ✅ Easy to modify Vehicle class—changes will automatically apply to Car, improving maintainability.

**9.** **Describe polymorphism. How does it contribute to flexibility and extensibility in software design? Give examples of function/operator overloading and function overriding.**

Polymorphism is an OOP principle that allows objects to be treated as instances of their parent class, enabling a single interface to be used for different implementations. This improves **flexibility** and **extensibility** in software design.

### **How Polymorphism Enhances Flexibility & Extensibility?**

✅ **Unified Interfaces** – Allows different implementations to be used interchangeably. ✅ **Code Reusability** – Reduces the need for separate functions for similar operations. ✅ **Easier Maintenance & Scalability** – New behaviors can be added without modifying existing code.

### **Types of Polymorphism in C++**

#### **1. Function Overloading (Compile-Time Polymorphism)**

Function overloading allows multiple functions with the same name but different parameters, enhancing flexibility.

cpp

#include <iostream>

using namespace std;

class MathOperations {

public:

int add(int a, int b) {

return a + b;

}

double add(double a, double b) { // Overloaded function

return a + b;

}

};

int main() {

MathOperations math;

cout << "Integer Addition: " << math.add(5, 3) << endl;

cout << "Double Addition: " << math.add(5.5, 3.2) << endl;

return 0;

}

✅ The same add() function works for both **integers** and **doubles**, demonstrating polymorphism.

#### **2. Operator Overloading (Compile-Time Polymorphism)**

Operator overloading allows custom behavior for operators like +, -, \*, enhancing flexibility.

cpp

#include <iostream>

using namespace std;

class Complex {

public:

int real, imag;

Complex(int r, int i) {

real = r;

imag = i;

}

Complex operator+(const Complex &obj) { // Overloading + operator

return Complex(real + obj.real, imag + obj.imag);

}

void display() {

cout << real << " + " << imag << "i" << endl;

}

};

int main() {

Complex c1(2, 3), c2(4, 5);

Complex sum = c1 + c2; // Uses overloaded + operator

sum.display();

return 0;

}

✅ The + operator is overloaded to add **complex numbers**, making it more intuitive.

#### **3. Function Overriding (Run-Time Polymorphism)**

Function overriding allows a derived class to modify a method from its base class, improving extensibility.

cpp

#include <iostream>

using namespace std;

class Animal {

public:

virtual void makeSound() { // Base class method

cout << "Animal makes a sound" << endl;

}

};

class Dog : public Animal {

public:

void makeSound() override { // Overridden method

cout << "Dog barks!" << endl;

}

};

int main() {

Animal \*ptr;

Dog myDog;

ptr = &myDog;

ptr->makeSound(); // Calls overridden function

return 0;

}

✅ Dog overrides makeSound(), demonstrating **dynamic polymorphism** using virtual functions.

**10. Explain the difference between "overloading" and "overriding".**

| **Aspect** | **Overloading (Compile-Time Polymorphism)** | **Overriding (Run-Time Polymorphism)** |
| --- | --- | --- |
| **Definition** | Defining multiple functions with the same name but **different parameter lists** in the same class. | Redefining an inherited function in a child class to **modify the behavior** of the parent class function. |
| **Purpose** | Provides multiple ways to execute a function using different inputs, improving **flexibility**. | Allows a child class to **customize** an inherited method, promoting **extensibility**. |

### ****2. Scope & Location****

| **Aspect** | **Overloading** | **Overriding** |
| --- | --- | --- |
| **Where it Occurs** | Within the **same class** (method signature differs). | Between **parent and child classes** (same method signature). |
| **Binding Time** | Resolved at **compile time** (early binding). | Resolved at **run time** (late binding). |

### ****3. Rules & Conditions****

| **Aspect** | **Overloading** | **Overriding** |
| --- | --- | --- |
| **Method Signature** | Must differ in the **number**, **type**, or **order** of parameters. | Must have the **same** method signature (name, parameters, return type). |
| **Modifiers Used** | No special keyword is required. | Uses the virtual keyword in C++ for dynamic binding. |
| **Return Type** | Can be different for overloaded methods. | Should be **same or more specific** than the parent class method. |

**11. List at least three advantages of using OOP in software development.**

### **Advantages of Object-Oriented Programming (OOP) in Software Development**

OOP provides several benefits that improve software design, maintainability, and scalability. Here are three major advantages:

1. **Encapsulation Enhances Security & Data Integrity**
   * OOP **encapsulates data** within objects, preventing unauthorized access and unintended modifications.
   * Reduces the risk of accidental changes by restricting direct interaction with data.
   * Example: In a banking system, an Account class ensures sensitive details (like balance) are accessed through controlled methods (getBalance() instead of modifying variables directly).
2. **Code Reusability Through Inheritance**
   * Inheritance allows child classes to reuse properties and behaviors from parent classes, **reducing redundant code**.
   * Enhances efficiency in large-scale applications where common features can be shared across multiple classes.
   * Example: A Vehicle class can define attributes like wheels and engine, and Car and Bike classes can inherit these without rewriting code.
3. **Improved Scalability & Maintainability**
   * OOP **organizes code into modular classes**, making it easier to modify, extend, and debug.
   * Changes in the parent class automatically apply to derived classes, simplifying software updates.
   * Example: If a Shape class has a draw() method, adding a new shape like Hexagon requires only extending the existing class rather than modifying every function separately.

**12. Give examples of application domains where OOP is commonly used (e.g., GUI development, game programming, etc.).**

OOP is widely used across various industries because of its modularity, scalability, and maintainability. Here are some key domains where OOP is commonly applied:

#### **1. GUI Development (Graphical User Interfaces)**

OOP is heavily used in creating GUI-based applications, where elements like buttons, text fields, and windows are treated as objects.

* **Example:** Java Swing, Tkinter (Python), and Qt (C++) use OOP to manage GUI components efficiently.

#### **2. Game Development**

OOP helps in designing games by structuring elements like characters, weapons, environments, and behaviors into modular objects.

* **Example:** Unity (C#), Unreal Engine (C++), and Godot (Python/GDScript) use OOP to handle game objects such as players, AI, physics, and animations.

#### **3. Web Development**

OOP supports backend development through frameworks that handle authentication, database connections, and business logic.

* **Example:** Django (Python), Laravel (PHP), and ASP.NET (C#) use OOP principles for building scalable and maintainable web applications.

#### **4. Mobile App Development**

OOP is essential in mobile applications, where different components (UI elements, activities, services) are modeled as objects.

* **Example:** Android development with Java/Kotlin, iOS development with Swift.

#### **5. Database Management Systems (DBMS)**

OOP aids in creating modular and efficient database architectures where tables, queries, and transactions are handled as objects.

* **Example:** PostgreSQL and MongoDB use OOP-based structures for efficient data modeling.

#### **6. Artificial Intelligence & Machine Learning**

OOP helps structure machine learning models and data handling efficiently, ensuring reusable code structures.

* **Example:** TensorFlow and PyTorch (Python) implement OOP for model training, deployment, and optimization.

#### **7. Embedded Systems & Robotics**

In embedded systems, OOP allows efficient management of hardware modules (sensors, motors, controllers) through object-oriented architecture.

* **Example:** Arduino (C++), ROS (Robot Operating System) uses OOP for modular robotics programming.

#### **8. Financial Systems & Banking Applications**

OOP is used to model complex financial transactions, customer accounts, and security mechanisms.

* **Example:** Banking software built with Java or C++ ensures scalable financial management

**13. Discuss the impact of OOP on code maintainability and reusability.**

Object-Oriented Programming (OOP) significantly enhances both code maintainability and reusability by organizing software design around objects and classes. Here's how:

**Impact on Maintainability:**

1. Encapsulation**:**
   * OOP allows bundling of data and methods that operate on that data into a single unit (a class). This encapsulation hides internal implementation details, reducing complexity and making it easier to understand and modify code without affecting other parts of the system.
2. Modularity:
   * By breaking down systems into discrete, self-contained classes, OOP promotes modular design. Developers can maintain or update individual modules without rewriting the entire codebase.
3. Ease of Debugging and Testing:
   * Encapsulated objects can be tested in isolation, making unit testing more effective. This simplifies debugging and helps in maintaining high code quality over time.
4. Code Scalability**:**
   * OOP systems are easier to scale as new features can often be added by extending existing classes, adhering to principles like Open/Closed Principle (open for extension, closed for modification).

**Impact on Reusability:**

1. Inheritance:
   * OOP supports inheritance, allowing new classes to be built upon existing ones. This encourages reuse of established code and reduces redundancy.
2. Polymorphism:
   * Through polymorphism, different classes can be treated as instances of the same interface or superclass. This allows code to be written more generically, increasing flexibility and reuse across different components.
3. Code Libraries and Frameworks:
   * Well-designed OOP code can be packaged into reusable libraries and frameworks that can be applied across multiple projects.
4. Design Patterns**:**
   * OOP facilitates the implementation of widely accepted design patterns (like Singleton, Factory, Observer), which promote reusable solutions to common programming problems

**14. How does OOP contribute to the development of large and complex software systems?**

Object-Oriented Programming (OOP) contributes significantly to the development of large and complex software systems by promoting better organization, maintainability, and scalability. Here’s how:

1. **Modularity**: OOP organizes code into **classes and objects**, allowing developers to break down complex systems into smaller, manageable parts. Each class encapsulates data and behavior, making it easier to understand and work on independently.
2. **Encapsulation**: By hiding internal state and requiring all interaction through well-defined interfaces (methods), OOP protects the integrity of data and reduces unintended side effects. This leads to more robust and secure systems.
3. **Inheritance**: Code reuse is made easier through inheritance, where new classes can inherit attributes and behaviors from existing ones. This reduces duplication and supports the creation of hierarchies that reflect real-world relationships.
4. **Polymorphism**: Polymorphism allows different classes to be treated through a common interface. This makes it easier to extend and modify parts of the system without changing the entire codebase, improving flexibility and scalability.
5. **Maintainability and Collaboration**: Since OOP promotes clear boundaries between components, large teams can work on different parts of the system simultaneously. It also simplifies debugging, testing, and updating specific components without affecting others.
6. **Design Patterns and Frameworks**: OOP supports common design patterns (e.g., Singleton, Factory, Observer) that provide proven solutions to common problems, speeding up development and enhancing code quality.

**15**. **Explain the benefits of using OOP in software development.**

**1**. Modularity

* Code is organized into classes and objects, each representing specific functionality.
* This makes it easier to divide work among team members and manage large codebases.

**2**. Reusability

* Through inheritance, existing code can be reused in new applications with little or no modification.
* Reduces duplication and development time.

**3.** Encapsulation

* Hides the internal state of objects and exposes only what is necessary through public methods.
* This protects data integrity and simplifies debugging.

**4.** Abstraction

* Allows developers to focus on what an object does instead of how it does it.
* Simplifies complex systems by exposing only essential features.

**5**. Polymorphism

* Enables objects of different types to be treated uniformly through a common interface.
* Promotes flexibility and scalability in code.

**6.** Maintainability

* Easier to update or fix specific parts of the system without affecting the entire application.
* Clear structure helps identify and isolate bugs quickly.

**7.** Scalability

* OOP makes it easier to scale projects by allowing new classes and features to be added with minimal disruption.

**8.** Better Collaboration

* Team members can work on different classes or modules independently.
* OOP encourages clear boundaries and responsibilities within code

**16.** **Describe the basic structure of a C++ program. What are the essential components?**

**#include <iostream> // Preprocessor directive**

**using namespace std; // Use the standard namespace**

**int main() { // Main function**

**int number = 10; // Variable declaration**

**cout << "Hello, C++! Number = " << number << endl; // Output**

**return 0; // Exit the program**

**}**

**Essential components:**

🔹 1. Preprocessor Directives

* These are instructions given to the compiler before the actual compilation begins.
* Most common is #include, which is used to include standard or user-defined header files.
* Example:  
  #include <iostream> – includes the input/output stream library for using cin and cout.

🔹 2. Namespace Declaration

* C++ uses namespaces to organize code into logical groups and to prevent name conflicts.
* The standard C++ library functions are in the std namespace.
* Example:  
  using namespace std; – allows direct use of cout, cin, etc., instead of std::cout.

🔹 3. Main Function

* This is the starting point of every C++ program.
* The function must return an int and may include a return 0; statement to indicate successful execution.
* Syntax:

cpp

CopyEdit

int main() {

// program logic

return 0;

}

🔹 4. Statements and Expressions

* These are the instructions that define what the program does.
* Examples include variable declarations, arithmetic operations, and function calls.
* Example:  
  int a = 10; or cout << "Hello";

🔹 5. Comments

* Not executed by the compiler, but used to explain the code.
* Two types:
  + Single-line: // comment
  + Multi-line: /\* comment \*/

🔹 6. Functions (Optional in small programs)

* For larger or modular programs, you define your own functions outside main() to handle specific tasks.
* Example:

cpp

CopyEdit

int add(int x, int y) {

return x + y;

}

**17. Explain the purpose of namespaces in C++. How do they help to avoid naming conflicts?**

**Namespaces** in C++ are used to group **identifiers** (like variables, functions, classes) under a named scope to **avoid naming conflicts**—especially in large programs or when using multiple libraries.

### 🔹 **Why Are Namespaces Needed?**

In large projects or when combining code from different sources (e.g., third-party libraries), you may encounter identifiers with the **same name**. Without namespaces, this leads to **naming collisions** and compiler errors.

#### Example (Without Namespace):

cpp

CopyEdit

int value = 10; // First declaration

int value = 20; // Error: Redefinition

### 🔹 **How Namespaces Help**

Namespaces allow you to **define the same name** in different scopes without conflict.

#### Example (With Namespace):

cpp

CopyEdit

namespace A {

int value = 10;

}

namespace B {

int value = 20;

}

int main() {

cout << A::value << endl; // Outputs 10

cout << B::value << endl; // Outputs 20

return 0;

}

Now both A::value and B::value can coexist peacefully.

**18.What are identifiers in C++? What rules must be followed when creating them?**

**Identifiers** in C++ are the **names** used to identify **variables, functions, arrays, classes, objects**, and other user-defined elements in a program.

They are fundamental to writing any C++ program because they allow you to **refer to stored data or operations** with readable, meaningful names.

### 🔹 **Examples of Identifiers:**

cpp

CopyEdit

int age;

float salary;

void displayInfo() { }

Here, age, salary, and displayInfo are identifiers.

### ✅ **Rules for Creating Identifiers in C++:**

| **Rule No.** | **Rule Description** |
| --- | --- |
| 1 | Identifiers can contain **letters (A-Z, a-z), digits (0-9), and underscores (\_) only.** |
| 2 | Identifiers **must begin with a letter or underscore, not** a digit. |
| 3 | Identifiers **cannot be the same as C++ reserved keywords** (e.g., int, return, while). |
| 4 | Identifiers are **case-sensitive** (Score, score, and SCORE are all different). |
| 5 | No special characters allowed (e.g., @, #, !, $, etc.). |
| 6 | There is **no length limit**, but very long identifiers can reduce readability. |

**19. What are the differences between variables and constants in C++? How are they declared?**

| **Feature** | **Variable** | **Constant** |
| --- | --- | --- |
| **Definition** | A storage location whose value **can change** during program execution. | A storage location whose value **cannot be changed** once defined. |
| **Mutability** | Mutable (can be modified) | Immutable (read-only after initialization) |
| **Declaration** | Regular data type declaration | Uses const keyword or #define pre-processor |
| **Usage** | For values that may vary (e.g., user input, calculations) | For fixed values (e.g., Pi, tax rate) |

### ✅ **How to Declare Variables in C++**

A variable stores a value that can change.

cpp

CopyEdit

int age = 25;

float salary = 50000.0;

char grade = 'A';

You can also declare them without assigning a value:

cpp

CopyEdit

int age;

### ✅ **How to Declare Constants in C++**

#### 🔹 Method 1: Using const keyword (preferred)

cpp

CopyEdit

const float PI = 3.14159;

const int MAX\_USERS = 100;

#### 🔹 Method 2: Using #define preprocessor (older style)

cpp

CopyEdit

#define PI 3.14159

#define MAX\_USERS 100

Note: const is type-safe and respects scope, while #define is a simple text substitution and does not.

**20. Explain how to use control structures (e.g., if-else, for, while) to control the flow of execution in a C++ program. Provide a simple code example**

**✅ Control Structures in C++**

**Control structures** are used to manage the **flow of execution** in a program based on conditions and loops. The main types are:

**🔹 1. if-else Statement**

Used to execute code based on a condition.

cpp

CopyEdit

if (condition) {

// Code if condition is true

} else {

// Code if condition is false

}

**🔹 2. for Loop**

Used when the number of iterations is known.

cpp

CopyEdit

for (int i = 0; i < 5; i++) {

cout << i << " ";

}

**🔹 3. while Loop**

Used when the number of iterations is **not** known in advance.

cpp

CopyEdit

int i = 0;

while (i < 5) {

cout << i << " ";

i++;

}

**✅ Simple Code Example Using All Three:**

cpp

CopyEdit

#include <iostream>

using namespace std;

int main() {

int number;

cout << "Enter a number: ";

cin >> number;

// if-else example

if (number % 2 == 0) {

cout << "Even number" << endl;

} else {

cout << "Odd number" << endl;

}

// for loop example

cout << "Counting from 1 to 5 using for loop:" << endl;

for (int i = 1; i <= 5; i++) {

cout << i << " ";

}

cout << endl;

// while loop example

cout << "Counting from 1 to 5 using while loop:" << endl;

int i = 1;

while (i <= 5) {

cout << i << " ";

i++;

}

return 0;

}

This program:

* Uses if-else to check if a number is even or odd.
* Uses a for loop and a while loop to print numbers 1 to 5.