Assignment-1

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

The **fundamental difference** between **procedural** and **object-oriented programming (OOP)** paradigms lies in **how they organize and manage code**:

* **Procedural Programming** focuses on **functions and procedures**. Code is organized as a sequence of instructions, and data is usually passed between functions.
* **Object-Oriented Programming** organizes code around **objects**—which combine **data (attributes)** and **functions (methods)** into reusable structures called **classes**.

**Example:**

**Procedural Programming (C-style):**

C

// Procedural: C language

#include <stdio.h>

void displayStudent(char name[], int age) {

printf("Name: %s, Age: %d\n", name, age);

}

int main() {

char name[] = "Pooja";

int age = 20;

displayStudent(name, age);

return 0;

}

// OOP: C++ language

#include <iostream>

using namespace std;

class Student {

public:

string name;

int age;

void display() {

cout << "Name: " << name << ", Age: " << age << endl;

}

};

int main() {

Student s;

s.name = "Pooja";

s.age = 20;

s.display();

return 0;

}

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

Ans-**Object-Oriented Programming (OOP)** is a programming paradigm based on the concept of **“objects”**, which are instances of **classes**. Each object contains **data** (attributes) and **methods** (functions) that operate on that data. OOP allows for organizing complex programs using reusable and modular components.

**Core Characteristics of OOP:**

1. **Encapsulation**
   * Wrapping data and methods into a single unit (class).
   * Keeps internal data safe from outside interference.
   * Example: A class Car with private attributes like speed and public methods to access or modify it.
2. **Abstraction**
   * Hiding complex implementation details and showing only the necessary features.
   * Example: A user interacts with a "drive()" method, without knowing how the engine or gears work.
3. **Inheritance**
   * Allows one class (child) to inherit properties and methods from another (parent).
   * Promotes code reusability.
   * Example: Class Dog inherits from class Animal.
4. **Polymorphism**
   * Means “many forms”. Objects can take on many forms depending on the context.
   * Achieved through **method overloading** and **method overriding**.
   * Example: A method makeSound() behaves differently for Dog and Cat classes

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

Ans-**Abstraction** in Object-Oriented Programming (OOP) refers to the concept of **hiding complex implementation details** and showing only the **essential features** of an object or system to the user.

In simpler terms, abstraction allows you to focus on **what an object does** instead of **how it does it**.

**Example:**

Think of a **car**:

* You use the **steering wheel, pedals, and gear**, but you don't need to know **how the engine works** internally.
* In OOP, a Car class might expose methods like start(), accelerate(), and brake()—but hides the internal code that controls the engine.

**Why Abstraction is Important:**

1. **Simplifies complexity**: Developers can use objects without knowing internal logic.
2. **Improves code maintainability**: Changes in implementation don’t affect users of the object.
3. **Enhances security**: Sensitive or unnecessary internal details are hidden.
4. **Supports modular design**: Code becomes easier to organize and understand.

4.What are the benefits of using OOP over procedural programming.

Ans-

| **Feature** | **OOP** | **Procedural Programming** |
| --- | --- | --- |
| **1. Code Reusability** | Encourages reuse through **classes** and **inheritance**. | Less reusable—requires copying and modifying code. |
| **2. Modularity** | Programs are divided into **objects**, making them modular and easier to manage. | Code is written as a sequence of procedures or functions—less organized. |
| **3. Abstraction** | Allows **hiding internal details** and exposing only functionality. | Generally lacks strong abstraction mechanisms. |
| **4. Encapsulation** | **Data and methods** are bundled together, providing better **security and integrity**. | Data is often exposed and shared freely, increasing the risk of accidental changes. |
| **5. Inheritance** | Promotes **code reuse** by allowing new classes to use properties of existing ones. | No built-in inheritance; code reuse must be manual. |
| **6. Polymorphism** | Enables writing **flexible and extendable code**—same method name behaves differently in different classes. | No support for polymorphism—requires workarounds. |
| **7. Easier Maintenance** | Changes in one part (class) usually don’t affect others—**low coupling**. | Tightly coupled code can break other parts when modified. |
| **8. Real-World Modeling** | Mimics **real-world entities**, making the design intuitive. | Doesn’t model the real world naturally. |

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

Ans-**eal-World Example: Library Management System**

**Why OOP is Well-Suited:**

In a **library system**, you deal with real-world entities like:

* **Books**
* **Members**
* **Librarians**
* **Transactions** (like issuing or returning books)

Each of these can be represented as a **class** in OOP, with specific **properties (data)** and **methods (behavior)**.

**Example Breakdown:**

* **Class: Book**
  + Properties: title, author, ISBN, isAvailable
  + Methods: borrow(), returnBook()
* **Class: Member**
  + Properties: name, memberID, borrowedBooks
  + Methods: borrowBook(), returnBook()
* **Class: Librarian**
  + Properties: employeeID, name
  + Methods: addBook(), removeBook()
* **Class: Transaction**
  + Properties: transactionID, book, member, dateIssued, dateReturned

**Why OOP Works Well:**

1. **Modularity**: Each class handles its own data and functions.
2. **Reusability**: Code for Book or Member can be reused in other projects (like a school library system).
3. **Scalability**: New features (e.g., digital books) can be added with minimal changes.

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

1. Ans-Define the four key principles of OOP: Encapsulation, Inheritance, Polymorphism, and Abstraction.

Ans-**1. Encapsulation**

**Bundling data and functions together; restricting direct access to some parts.**

cpp

#include <iostream>

using namespace std;

class Student {

private:

int age; // private data

public:

void setAge(int a) {

if (a > 0)

age = a;

}

int getAge() {

return age;

}

};

int main() {

Student s;

s.setAge(20);

cout << "Age: " << s.getAge() << endl;

return 0;

}

**2. Inheritance**

**One class inherits features from another.**

cpp

#include <iostream>

using namespace std;

class Animal {

public:

void sound() {

cout << "Some generic animal sound" << endl;

}

};

class Dog : public Animal {

public:

void bark() {

cout << "Dog barks" << endl;

}

};

int main() {

Dog d;

d.sound(); // inherited

d.bark(); // own method

return 0;

}

**3. Polymorphism**

**Same function behaves differently in different classes (runtime polymorphism).**

cpp

#include <iostream>

using namespace std;

class Shape {

public:

virtual void draw() {

cout << "Drawing a shape" << endl;

}

};

class Circle : public Shape {

public:

void draw() override {

cout << "Drawing a circle" << endl;

}

};

int main() {

Shape\* s;

Circle c;

s = &c;

s->draw(); // Calls Circle's draw() because of virtual function

return 0;

}

**4. Abstraction**

**Hiding internal details; showing only necessary features.**

cpp

#include <iostream>

using namespace std;

class Vehicle {

public:

virtual void move() = 0; // Pure virtual function (abstract method)

};

class Car : public Vehicle {

public:

void move() override {

cout << "Car is moving on the road" << endl;

}

};

int main() {

Vehicle\* v = new Car();

v->move(); // Only essential behavior is shown

delete v;

return 0;

}

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

Ans-Encapsulation in OOP: How It Protects Data and Creates Modular Code

Encapsulation is the practice of keeping **data (variables)** and the **methods** that operate on that data within one unit (usually a class), and **restricting access** to that data from outside the class.

**How Encapsulation Protects Data:**

1. **Private members** prevent direct access from outside the class.
2. Access is only allowed through **public methods** (getters and setters).
3. You can add **validation or logic** before modifying the data.

**How Encapsulation Creates Modular Code:**

1. Each class handles its own **data and functionality**.
2. Changes in one class don’t affect others if interfaces remain the same.
3. Makes the code **easy to maintain**, **test**, and **reuse**.

**C++ Example: Encapsulation**

cpp

CopyEdit

#include <iostream>

using namespace std;

class BankAccount {

private:

double balance; // Private data, not accessible directly

public:

// Constructor

BankAccount() {

balance = 0.0;

}

// Public method to deposit money

void deposit(double amount) {

if (amount > 0)

balance += amount;

}

// Public method to get balance

double getBalance() {

return balance;

}

};

int main() {

BankAccount acc;

acc.deposit(500.0); // Safe access through method

cout << "Balance: " << acc.getBalance() << endl;

// acc.balance = 1000; // Error: 'balance' is private

return 0;

}

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

Ans-**Inheritance** is a key concept in Object-Oriented Programming where a **class (child or derived class)** can inherit **properties and behaviors (methods)** from another **class (parent or base class)**.

**How Inheritance Helps:**

**Code Reuse:**

* You don’t need to rewrite code that is already defined in the parent class.
* Shared functionality is written once and reused across multiple classes.

**Maintainability:**

* Changes in the parent class automatically reflect in all child classes (if applicable).
* Makes the code easier to update and extend.

**🔹 C++ Example: Inheritance**

cpp

CopyEdit

#include <iostream>

using namespace std;

// Base class

class Animal {

public:

void eat() {

cout << "This animal eats food." << endl;

}

};

// Derived class

class Dog : public Animal {

public:

void bark() {

cout << "The dog barks." << endl;

}

};

int main() {

Dog d;

d.eat(); // Inherited from Animal

d.bark(); // Dog's own method

return 0;

}

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

Ans-**Polymorphism** means **"many forms"**. It allows the same function, operator, or method name to **behave differently** depending on the context—**class type or number/type of arguments**.

**Types of Polymorphism in C++:**

| **Type** | **Description** | **Example** |
| --- | --- | --- |
| **Compile-time** | Known at compile time | Function Overloading, Operator Overloading |
| **Run-time** | Resolved at runtime | Function Overriding (with inheritance & virtual functions) |

**🔹 1. Function Overloading (Compile-time Polymorphism)**

Same function name with **different parameters**.

cpp

CopyEdit

#include <iostream>

using namespace std;

class Print {

public:

void show(int a) {

cout << "Integer: " << a << endl;

}

void show(string s) {

cout << "String: " << s << endl;

}

};

int main() {

Print p;

p.show(10);

p.show("Hello");

return 0;

}

**🔹 2. Operator Overloading (Compile-time Polymorphism)**

You can define how an operator works with **user-defined objects**.

cpp

CopyEdit

#include <iostream>

using namespace std;

class Complex {

public:

int real, imag;

Complex(int r = 0, int i = 0) {

real = r; imag = i;

}

// Overload '+' operator

Complex operator + (const Complex& obj) {

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 c3 = c1 + c2; // Uses overloaded operator

c3.display();

return 0;

}

**3. Function Overriding (Run-time Polymorphism)**

Same method name in **base and derived class**. Use **virtual function** to allow dynamic binding.

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

Ans-

| **Feature** | **Overloading** | **Overriding** |
| --- | --- | --- |
| **Definition** | Defining **multiple functions with the same name** but **different parameters** in the **same scope**. | Redefining a **base class method** in the **derived class** with the **same signature**. |
| **Type** | **Compile-time Polymorphism** | **Run-time Polymorphism** |
| **Scope** | Occurs **within the same class** | Occurs **between base and derived classes** |
| **Method Signature** | Must have **different number or types** of parameters | Must have the **same name and parameters** |
| **Keyword Needed?** | No special keyword needed | Use virtual in base class and override (optional) in derived class |
| **Purpose** | Increase **flexibility** by allowing multiple ways to call a method | Allows **custom behavior** for base class methods |

**Example of Function Overloading:**

cpp

CopyEdit

class Print {

public:

void show(int a) {

cout << "Integer: " << a << endl;

}

void show(string s) {

cout << "String: " << s << endl;

}

};

**Example of Function Overriding:**

cpp

CopyEdit

class Animal {

public:

virtual void sound() {

cout << "Some generic sound" << endl;

}

};

class Dog : public Animal {

public:

void sound() override {

cout << "Bark" << endl;

}

};

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

Ans-1. Code Reusability

* OOP allows you to reuse existing classes through **inheritance**.
* You can build new features by extending old code, reducing duplication.

*Example*: A Vehicle class can be reused for Car, Bike, and Bus by inheriting it.

**2. Modularity and Maintainability**

* Each class is a **self-contained module** with specific responsibility.
* Easier to debug, test, and update parts of the program **without affecting others**.

*Example*: Fixing a bug in the Payment class won't affect the User or Product classes.

**3. Data Protection (Encapsulation)**

* Data is protected from direct access using **private members** and accessed through **public methods**.
* Ensures **controlled access** and increases **security**.

*Example*: User passwords in a class can only be changed or viewed through secure functions.

Bottom of Form

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

Ans- **GUI Development**: OOP is used to create components like buttons and text fields (e.g., Java Swing, Qt).

**Game Programming**: Models objects like players, enemies, and weapons (e.g., Unity, Unreal Engine).

**Web Development**: Manages entities like users, products, and orders (e.g., Django, Spring).

**Embedded Systems**: Used for devices like sensors and actuators.

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

1. Ans-Maintainability:
   * Modular Design: Breaks the code into self-contained classes, making it easier to maintain.
   * Encapsulation: Protects internal class details, allowing changes without affecting other parts of the system.
   * Easier Updates: Changes to one class don't impact others, reducing bugs.
2. Reusability:
   * Inheritance: Enables reuse of existing functionality across different classes.
   * Polymorphism: Allows the same method to behave differently based on the object type.
   * Modular Components: Promotes reusable code modules that can be applied in different projects.

Bottom of Form

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

1. Ans-**Modularity**: Breaks the system into manageable classes, each responsible for a specific functionality.
2. **Scalability**: Easily extendable with inheritance and polymorphism, allowing for new features without changing existing code.
3. **Reusability**: Promotes code reuse through inheritance, reducing redundancy and development time.
4. **Encapsulation**: Protects data and maintains integrity by restricting direct access to it.
5. **Maintainability**: Makes debugging and updating easier since changes in one class don’t affect others.
6. **Collaboration**: Enables teamwork by allowing different teams to work on different parts independently.
7. **Abstraction**: Hides complex details, allowing developers to focus on high-level functionality.

15. Explain the benefits of using OOP in software development.Top of Form

Ans-same as 14.

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

Ans-**1. Preprocessor Directives**

* **Purpose**: These lines are processed before compilation. They include header files, macros, and other instructions that provide information to the compiler.
* **Example**:

cpp

#include <iostream> // Include the standard input-output stream library

**2. Namespace Declaration**

* **Purpose**: The std namespace is often used to avoid having to write std:: before every standard library function or object.
* **Example**:

cpp

using namespace std; // Makes the standard library functions accessible without std:: prefix

**3. Main Function**

* **Purpose**: Every C++ program must have a main() function. This is the entry point where the program starts execution.
* **Example**:

cpp

int main() {

// Program code starts here

return 0; // End of the program, returning a success status

}

**4. Variable Declarations and Initialization**

* **Purpose**: Variables are declared and initialized to store data during the program's execution.
* **Example**:

cpp

int age = 25; // Declaring and initializing an integer variable

float salary = 50000.5; // Declaring a floating-point variable

**5. Statements and Expressions**

* **Purpose**: These are the instructions that the program executes. They can be simple assignments, function calls, loops, or conditionals.
* **Example**:

cpp

cout << "Hello, World!" << endl; // Output a message to the console

**6. Functions**

* **Purpose**: Functions allow code to be reused and organized into logical blocks. While the main() function is mandatory, additional functions can be created for specific tasks.
* **Example**:

cpp

void displayMessage() {

cout << "This is a function" << endl;

}

**7. Classes and Objects (Optional for OOP)**

* **Purpose**: In object-oriented C++ programs, classes are defined to model real-world entities. These classes contain attributes (variables) and methods (functions).
* **Example**:

cpp

class Person {

public:

string name;

int age;

void introduce() {

cout << "Hello, my name is " << name << " and I am " << age << " years old." << endl;

}

};

**8. Return Statement**

* **Purpose**: The return statement inside main() returns an exit status to the operating system. 0 typically indicates successful execution.
* **Example**:

cpp

return 0; // Exiting the main function with a success status

**Full Example of a Simple C++ Program:**

cpp

#include <iostream> // Include the header for input-output stream

using namespace std; // Use the standard namespace

// Function definition

void greet() {

cout << "Hello, World!" << endl; // Output a message to the console

}

// Main function

int main() {

greet(); // Calling the greet function

return 0; // End of the program

}

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

Ans-Namespaces in C++ are used to **organize code** and **prevent naming conflicts**. They help by:

1. **Avoiding Naming Conflicts**: They allow functions, variables, and classes with the same name to coexist in different namespaces.
2. **Organizing Code**: Related code can be grouped under a namespace, making large programs more manageable.
3. **Selective Access**: The **scope resolution operator** (::) is used to specify which function or class to access from a particular namespace.

**Example:**

cpp

CopyEdit

namespace Library1 {

void print() { std::cout << "Library1 print" << std::endl; }

}

namespace Library2 {

void print() { std::cout << "Library2 print" << std::endl; }

}

int main() {

Library1::print(); // Calls Library1's print

Library2::print(); // Calls Library2's print

return 0;

}

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

Ans-Identifiers are names used for variables, functions, classes, and other entities in C++.

**Rules:**

1. **First character**: Must be a letter (A-Z, a-z) or an underscore (\_ (not a digit)).
2. **Subsequent characters**: Can include letters, digits, and underscores.
3. **No reserved keywords**: Cannot use C++ keywords like int, return, etc.
4. **Case-sensitive**: age, Age, and AGE are different identifiers.
5. **Avoid starting with a digit**: 1variable is invalid.
6. **Meaningful names**: Keep identifiers meaningful for readability.
7. What are the differences between variables and constants in C++? How are they declared?

Ans-

| **Feature** | **Variables** | **Constants** |
| --- | --- | --- |
| **Definition** | A variable is a memory location that can store data and whose value can change during program execution. | A constant is a value that remains fixed throughout the program's execution. |
| **Value** | The value can be changed or modified during execution. | The value is fixed and cannot be changed after initialization. |
| **Declaration** | Declared using a data type, like int, float, etc., and the value can be assigned later. | Declared using the const keyword, followed by the data type and initialization. |
| **Example** | int age = 25; | const int MAX\_AGE = 100; |
| **Memory Allocation** | Memory is allocated, and the value can vary. | Memory is allocated, but the value is constant. |
| **Usage** | Used to store data that may change, such as counters, user input, etc. | Used for values that should not change, like PI, MAX\_SIZE, etc. |

**How to Declare Variables and Constants in C++**

1. **Declaring Variables**:
   * You declare variables with a data type followed by the variable name.
   * Example:

cpp

int age = 25; // Variable declaration

float salary = 5000.5;

1. **Declaring Constants**:
   * Constants are declared using the const keyword, followed by the data type and the value.
   * Example:

cpp

const int MAX\_AGE = 100; // Constant declaration

const float PI = 3.14159;

Bottom of Form

1. 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

Ans-**Control Structures in C++**

Control structures in C++ are used to determine the flow of execution in a program based on conditions or loops. The common control structures are:

1. **If-else**: Used for decision-making based on conditions.
2. **For loop**: Used for repeating a block of code a specific number of times.
3. **While loop**: Used for repeating a block of code while a condition is true.

**1. If-else Statement**

The if-else statement allows conditional execution of code based on whether a condition is true or false.

**Syntax:**

cpp

if (condition) {

// Code to execute if condition is true

} else {

// Code to execute if condition is false

}

**Example:**

cpp

#include <iostream>

using namespace std;

int main() {

int age = 18;

if (age >= 18) {

cout << "You are an adult." << endl;

} else {

cout << "You are a minor." << endl;

}

return 0;

}

**2. For Loop**

The for loop is used when you know in advance how many times you want to repeat a block of code.

**Syntax:**

cpp

for (initialization; condition; increment/decrement) {

// Code to execute repeatedly

}

**Example:**

cpp

#include <iostream>

using namespace std;

int main() {

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

cout << "Iteration " << i << endl;

}

return 0;

}

This loop prints "Iteration 1" to "Iteration 5" because i starts at 1 and increments until 5.

**3. While Loop**

The while loop executes a block of code as long as a given condition is true.

**Syntax:**

cpp

while (condition) {

// Code to execute while condition is true

}

**Example:**

cpp

#include <iostream>

using namespace std;

int main() {

int i = 1;

while (i <= 5) {

cout << "Iteration " << i << endl;

i++; // Increment i after each iteration

}

return 0;

}

Bottom of Form

15

Bottom of Form

Bottom of Form