**ASSIGNMENT 1**

Q1. What is the fundamental difference between procedural and object-oriented programming paradigms? Provide a brief example to illustrate.

Answer: The fundamental difference between procedural programming and object-oriented programming (OOP) lies in how they structure and manage code.

* Procedural Programming: Focuses on a sequence of procedures or functions that operate on data. It follows a top-down approach and is centered around functions.
* Object-Oriented Programming (OOP): Organizes code into objects, which encapsulate both data (attributes) and behaviour (methods). It follows a bottom-up approach and promotes concepts like encapsulation, inheritance, and polymorphism.

Difference:

|  |  |
| --- | --- |
| Procedural oriented programming | Object - oriented programming |
| In procedural programming, the program is divided into small parts called function. | In object-oriented programming, program is divided into small parts called object. |
| There is no access specifier. | It has access specifiers like public, private, protected etc |
| Did not provide real world mapping | Provide real world mapping. |
| It is less secure. | It is more secure as it provides data hiding. |
| Example: C, FORTAN etc. | Example: C++, Java, Python etc. |

Example:

**Procedural Approach ( C++)**

#include <iostream>

using namespace std;

struct Car {

string name;

int speed;

};

void displayCar(Car c) {

cout << "Car: " << c.name << ", Speed: " << c.speed << " km/h" << endl;

}

int main() {

Car myCar = {"Toyota", 120};

displayCar(myCar);

return 0;

}

Issues in Procedural Programming: Functions operate on separate data structures, leading to poor data security and maintainability.

**OOP Approach (Using Class in C++)**

#include <iostream>

using namespace std;

class Car {

private:

string name;

int speed;

public:

Car(string n, int s) : name(n), speed(s) {}

void display() {

cout << "Car: " << name << ", Speed: " << speed << " km/h" << endl;

}

};

int main() {

Car myCar("Toyota", 120);

myCar.display();

return 0;

}

Q2. Define Object -oriented programming(OOP).What are its core characteristics?

Answer: Definition: “Object-oriented programming as an approach that provides a way of modularizing programs by creating partitioned memory area for both data and functions that can be used as templates for creating copies of such modules on demand.”

Thus, an object is considered to be partitioned area of computer memory that stores data and set of operations that can access that data. Since the memory partitions are independent, the objects can be used in a variety of different programs without modifications.

Characteristics of Oriented Programming

• Emphasis is on data rather than procedure.

• Programs are divided into what are known as objects.

• Data structures are designed such that they characterize the objects.

• Functions that operate on the data of an object are tied together in the data structure.

• Data is hidden and cannot be accessed by external functions.

• Objects may communicate with each other through functions.

• New data and functions can be easily added whenever necessary.

• Follows bottom-up approach in program design.

Q3. Explain the concept of "abstraction" within the context of OOP. Why is it important?

Answer: Abstraction is the process of hiding implementation details while exposing only relevant functionality to the user. It helps in reducing complexity and increasing maintainability.

Example of Abstraction in C++

#include <iostream>

using namespace std;

class Car {

public:

virtual void startEngine() = 0; // Abstract method

};

class Tesla : public Car {

public:

void startEngine() override {

cout << "Tesla engine started!" << endl;

} };

int main() {

Car\* myCar = new Tesla();

myCar->startEngine();

delete myCar;

return 0;

}

The startEngine() function is abstract (pure virtual function).

The Tesla class implements the function, providing its own behavior.

Importance of Abstraction:

* Simplifies program design.
* Hides unnecessary details from users.
* Enhances security by preventing access to internal implementation.

Q4. What are the benefits of using OOPs over procedural programming?

Answer **Benefits of OOP over procedural programming are:**

* **Encapsulation**: One of the key concepts of OOP is encapsulation, which allows data to be hidden and protected from unauthorized access. This helps to prevent errors, improves security, and makes code easier to maintain. Example — Suppose you are building a program to manage user data, and you want to ensure that the user’s password is hidden from other parts of the program. Using OOP, you can encapsulate the password data within a class and only allow authorized code to access it.
* **Abstraction:** OOP allows developers to model complex systems in a simplified manner, making it easier to understand, modify, and extend the code. Abstraction also allows the code to be modularized, making it easier to reuse and test. Example — Let’s say you are building a game and you want to represent a player’s attributes, such as health and strength. Instead of writing separate code for each attribute, you can abstract them into a single class that represents the player’s overall state.
* **Inheritance**: In OOP, classes can inherit properties and methods from other classes, reducing the amount of code that needs to be written. This helps to reduce code duplication and improve code readability. Example — Consider a program that represents different types of vehicles, such as cars and trucks. Instead of writing separate code for each type of vehicle, you can create a parent class for “Vehicle” and then create child classes for “Car” and “Truck” that inherit properties and methods from the parent class.
* **Polymorphism**: OOP allows for polymorphism, which means that objects can be used in different ways depending on the context. This improves code flexibility and makes it easier to write code that is reusable and extensible. Example — Imagine a program that needs to calculate the area of different shapes, such as squares and circles. Using OOP, you can create a parent class for “Shape” and then create child classes for “Square” and “Circle” that each implement a unique “area” method.
* **Code Reusability**: OOP promotes code reusability by allowing developers to create classes and objects that can be reused in different parts of the code. This saves time and effort, and also makes it easier to maintain the code over time. Example — Suppose you are building a program that needs to access data from a database. Using OOP, you can create a class that handles the database connections and queries, and then reuse this class in different parts of the program to access and manipulate the data.

Q5. Give a real-world example of a problem that is well-suited to be solved using an OOP approach. Explain why.

Ans A banking system is well-suited for OOP because it involves multiple entities like customers, accounts, and transactions, which can be represented as classes and objects.

Why OOP?

* Encapsulation: Protects sensitive data (e.g., balance) using private members.
* Inheritance: Savings Account and Current Account can inherit from Bank Account.
* Polymorphism: Different transactions (deposit, withdraw) share a common interface.
* Modularity & Scalability: New features can be added easily without modifying existing code.

C++ Example:

class Bank Account {

private:

double balance;

public:

Bank Account(double initial Balance) { balance = initial Balance; }

void deposit(double amount) { balance += amount; }

void withdraw(double amount) { if (amount <= balance) balance -= amount; }

};

Q6. Define the four key principles of OOP: Encapsulation, Inheritance, Polymorphism, and Abstraction.

Answer Four Key Principles of OOP

* **Encapsulation** – The process of hiding data within a class and restricting direct access to it. Data can only be accessed through public methods, ensuring data security and integrity.
* **Inheritance** – Allows a child class to inherit attributes and methods from a parent class, promoting code reuse and hierarchical relationships.
* **Polymorphism –** Enables a function or method to take multiple forms, allowing function overloading and overriding to enhance flexibility and extensibility in software design.
* **Abstraction** – The technique of hiding complex implementation details and exposing only essential features, reducing complexity and improving maintainability.

Q7. Explain how encapsulation helps to protect data and create modular code. Give an example using a class and its members.

Answer: Encapsulation is one of the fundamental principles of Object-Oriented Programming (OOP). It helps protect data by restricting direct access and allows controlled modification through public methods. This ensures data security, integrity, and modularity, making the code more maintainable and reusable.

Example:

#include <iostream>

using namespace std;

class BankAccount {

private:

double balance; // Private data member

public:

BankAccount(double initialBalance) { balance = initialBalance; }

void deposit(double amount) { balance += amount; }

void withdraw(double amount) {

if (amount <= balance)

balance -= amount;

else

cout << "Insufficient balance!" << endl;

}

double getBalance() { return balance; } // Public method to access balance

};

int main() {

BankAccount acc(5000);

acc.deposit(2000);

acc.withdraw(1000);

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

return 0;

}

Explanation:

The balance variable is private, preventing direct access from outside the class. Public methods (deposit(), withdraw(), getBalance()) provide controlled access. This ensures data security, modularity, and maintainability.

Q8. What is inheritance? How does it promote code reuse and maintainability? Provide a simple example using classes.

Answer Inheritance is an OOP principle that allows a child class to acquire properties and behaviors from a parent class, reducing redundancy and improving maintainability.

How Inheritance Helps?

**Code Reuse** – Common attributes and methods are defined once in the parent class and reused in derived classes.

**Maintainability** – Changes made in the base class automatically apply to all derived classes, making updates easier.

Example:

#include <iostream>

using namespace std;

class Animal {

public:

void eat() { cout << "Eating..." << endl; }

};

class Dog : public Animal { // Inheriting from Animal

public:

void bark() { cout << "Barking..." << endl; }

};

int main() {

Dog d;

d.eat(); // Inherited method

d.bark();

return 0;

}

Q9. Describe polymorphism. How does it contribute to flexibility and extensibility in software design? Give examples of function/operator overloading and function overriding.

Answer Polymorphism in OOP

Polymorphism means "many forms", allowing functions or methods to behave differently based on the object or data type. It helps in writing flexible and extensible code.

Types of Polymorphism

* **Compile-time Polymorphism (Static Binding):** Achieved through Function Overloading and Operator Overloading.
* **Run-time Polymorphism (Dynamic Binding):** Achieved through Function Overriding.

**Function Overloading Example**

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

#include <iostream>

using namespace std;

class Math {

public:

int add(int a, int b) { return a + b; }

double add(double a, double b) { return a + b; }

};

int main() {

Math obj;

cout << obj.add(5, 3) << endl; // Calls int version

cout << obj.add(2.5, 3.2) << endl; // Calls double version

}

Operator Overloading Example

Operator overloading allows operators (like +, -, \*) to work with user-defined types.

#include <iostream>

using namespace std;

class Complex {

public:

int real, imag;

Complex(int r, int i) : real(r), imag(i) {}

Complex operator+(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; // Calls overloaded operator+

c3.display();

}

**Function Overriding Example**

Function overriding allows a subclass to provide a specific implementation of a function defined in its base class.

#include <iostream>

using namespace std;

class Parent {

public:

virtual void show() { cout << "Parent class" << endl; }

};

class Child : public Parent {

public:

void show() override { cout << "Child class" << endl; }

};

int main() {

Parent\* obj = new Child();

obj->show(); // Calls overridden method in Child class}

Q10. Explain the difference between "overloading" and "overriding".

Answer:

| **Feature** | **Overloading** | **Overriding** |
| --- | --- | --- |
| Definition | Multiple functions with the same name but different parameters | Redefining a base class function in the derived class |
| Type | Compile-time polymorphism | Run-time polymorphism |
| Classes Involved | Same class | Parent and child class |
| Example | add(int, int) & add(double, double) | show() function overridden in derived class |

Q11.List at least three advantages of using OOP in software development.

Answer Advantages of OOP

**Encapsulation** – Protects data by restricting direct access.

**Reusability** – Inheritance allows code to be reused, reducing redundancy.

**Scalability** – OOP makes it easier to build large-scale applications.

Q12. Give examples of application domains where OOP is commonly used (e.g., GUI development, game programming, etc.).

Answer Applications of OOP

**Graphical User Interface (GUI)**: OOP is heavily used in developing user interfaces for applications, such as Windows applications and Android apps.

**Game Development**: OOP helps in designing characters, items, events, and various components in gaming environments.

**Web Development**: OOP can be used to create dynamic web pages and applications. For example, frameworks like React are built on OOP principles.

**Mobile Applications**: OOP is used in developing iOS and Android applications to manage objects and data efficiently.

**Real-Time Systems**: Applications that require real-time data processing, such as simulation and modeling, use OOP to manage complex interactions and data flow.

Q13. Discuss the impact of OOP on code maintainability and reusability.

Answer **Impact of OOP on Maintainability & Reusability~**

**Maintainability**: Since OOP follows modularity, debugging and maintaining code becomes easier.

**Reusability:** Inheritance and polymorphism allow developers to reuse code instead of writing it from scratch.

Q14. How does OOP contribute to the development of large and complex software systems?

Ans **OOP in Large & Complex Software Systems**

* Divides software into independent objects that interact with each other.
* Encourages collaboration by allowing different teams to work on separate modules.
* Reduces development time by using reusable components.

Q15. Explain the benefits of using OOP in software development.

Ans Benefits of OOP in software devlopment

* Code Reusability – Inheritance reduces redundancy.
* Maintainability – Modular structure makes debugging easier.
* Scalability – Supports large and complex applications.
* Security – Encapsulation protects data.
* Flexibility – Polymorphism allows adaptable code.
* Better Collaboration – Teams can work independently.
* Faster Development – Code reuse speeds up development.

Q16. Describe the basic structure of a C++ program. What are the essential components?

Ans Basic Program Structure in C++:

Here's a basic overview of a simple C++ program structure:

#include <iostream> // Preprocessor directive

// Entry point of the program

int main() {

std::cout << "Hello, World!" << std::endl; // Output to console

return 0; // Exit status of the program

}

Key Components:

**Preprocessor Directives**: These are lines included at the beginning of the program, which instruct the compiler to include libraries (#include <iostream>).

**Main Function**: This is the entry point of every C++ program. The int main() function is where execution starts.

**Statements and Expressions**: Inside the main() function, you write code statements. For example, std::cout is used for output, followed by the << operator and the string to display.

**Return Statement**: return 0; signifies the end of the main() function and returns control to the operating system. 0 usually indicates the program ended successfully.

Q17. Explain the purpose of namespaces in C++. How do they help to avoid naming conflicts?

Ans Purpose of Namespaces in C++

A namespace in C++ is used to organize code and prevent naming conflicts, especially in large projects where multiple libraries may define the same function or variable names.

**Help Avoid Naming Conflicts**

* Without namespaces, if two libraries define a function with the same name, it would cause an error.
* Namespaces allow you to define the same function name in different scopes.

Example

#include <iostream>

namespace A {

int value = 10;

}

namespace B {

int value = 20;

}

int main() {

std::cout << A::value << std::endl; // Output: 10

std::cout << B::value << std::endl; // Output: 20

return 0;

}

Here, value is defined in two different namespaces, preventing conflicts.

Q18. What are identifiers in C++? What rules must be followed when creating them?

Ans **Identifiers in C++**

Identifiers are the names given to variables, functions, arrays, classes, etc., to uniquely identify them in a program.

Rules for Identifiers

* Must start with a letter (A-Z or a-z) or an underscore (\_)
* Cannot use C++ keywords as identifiers
* Must be case-sensitive (MyVar and myvar are different)
* Can contain letters, digits (0-9), and underscores

Examples

Valid: age, \_height, totalMarks

Invalid: 2days (cannot start with a number), int (keyword), total marks (space not allowed)

Q19. What are the differences between variables and constants in C++? How are they declared?

### Ans **Variables vs. Constants in C++**

| **Feature** | **Variable** | **Constant** |
| --- | --- | --- |
| Can change value? | Yes | No |
| Declaration | int x = 5; | const int x = 10; |
| Storage | Stored in RAM, can be updated | Stored in ROM (or immutable memory) |

**Example**

int age = 25; // Variable

const float PI = 3.14159; // Constant

Here, age can be changed, but PI remains fixed.

Q20. 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 manage the flow of execution in a program by making decisions and repeating code blocks. The main types are:

1. if-else (Decision Making)

Used to execute different code based on conditions.

Example:

#include <iostream>

using namespace std;

int main() {

int age;

cout << "Enter your age: ";

cin >> age;

if (age >= 18) {

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

} else {

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

}

return 0;

}

If age >= 18, it prints "You are an adult."

Otherwise, it prints "You are a minor."

2. for Loop (Fixed Iteration)

Executes a block of code a fixed number of times.

Example:

#include <iostream>

using namespace std;

int main() {

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

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

}

return 0;

}

Prints numbers 1 to 5 in sequence.

3. while Loop (Condition-Based Iteration)

Repeats a block while a condition is true.

Example:

#include <iostream>

using namespace std;

int main() {

int i = 1;

while (i <= 5) {

cout << "Number: " << i << endl;

i++; // Increment i

}

return 0;

}

Runs until i reaches 5, printing each number.