1. Object-oriented programming (OOP) is a programming style characterized by identifying classes of objects closely linked with the methods (functions) with which they are associated. It also includes ideas on the inheritance of attributes and methods.

Significance of OOP

• Modularity: Code is arranged into independent units, which enhances readability and structure.
• Reusability: Effective code reuse is made possible by inheritance
• Maintainability: Modifications made to one aspect of the code have little effect on the others.
• Versatility: facilitates the development of complex software systems.
• Security: Data is shielded from unauthorized access by encapsulation.

Encapsulation – Restricts direct access to data by bundling it with methods.

E.g. an ATM hiding banking operations.

Abstraction – Hides complex details and exposes only necessary functionality.

E.g. driving a car without knowing how the engine works.

Inheritance – Allows a class to inherit properties and behaviors from another.

E.g. like a child inheriting traits from parents.

Polymorphism – Enables the same action to behave differently based on context

E.g. a person acting as a student, customer, or player.

2. **Employee Data Management**: Store and manage employee details like name, ID, and salary. This includes adding, updating, and deleting employee records, and retrieving and displaying employee information.

   Search and Filter Functionality: Looks for workers using particular categories (e.g., name, ID)

   Data Security and Integrity: Ensure data accuracy and consistency. Implement access controls and user authentication to protect private information.

   User Interface: allows users to manage employee records and perform operations efficiently

b. **Encapsulation:** Employee data (name, ID, salary) should be **private** and accessible via **getter and setter methods**. Preventing direct data modification ensures security.

Inheritance: A base-class **Employee** can hold common attributes. Derived classes like **Manager** and **Engineer** can extend Employees with specific behaviors.

Polymorphism: Implement method overriding for salary calculation in different employee types. Use a common interface to ensure flexibility.

3.
```
#include <iostream>
#include <vector>
using namespace std;

// Base Class: Employee
class Employee {
protected:
    string name;
    int employeeID;
    double salary;

public:
    virtual void inputDetails() {
        cout << "Enter Name: ";
        cin >> ws;
        getline(cin, name);
        cout << "Enter Employee ID: ";
        cin >> employeeID;
        cout << "Enter Salary: ";
        cin >> salary;
    }

    virtual void displayDetails() {
```

```
        cout << "\nEmployee Details:" << endl;
        cout << "Name: " << name << endl;
        cout << "Employee ID: " << employeeID << endl;
        cout << "Salary: $" << salary << endl;
    }

    int getEmployeeID() { return employeeID; }
    virtual ~Employee() {}
};

// Derived Class: Manager
class Manager : public Employee {
private:
    string department;
    double bonus;

public:
    void inputDetails() override {
        Employee::inputDetails();
        cout << "Enter Department: ";
        cin >> ws;
        getline(cin, department);
        cout << "Enter Bonus: ";
        cin >> bonus;
    }
```

```
    void displayDetails() override {
        Employee::displayDetails();
        cout << "Department: " << department << endl;
        cout << "Bonus: ksh" << bonus << endl;
    }
};

// Derived Class: Engineer
class Engineer : public Employee {
private:
    string specialization;
    string projectAssigned;

public:
    void inputDetails() override {
        Employee::inputDetails();
        cout << "Enter Specialization: ";
        cin >> ws;
        getline(cin, specialization);
        cout << "Enter Project Assigned: ";
        getline(cin, projectAssigned);
    }

    void displayDetails() override {
        Employee::displayDetails();
        cout << "Specialization: " << specialization << endl;
```

```
        cout << "Project Assigned: " << projectAssigned << endl;
    }
};

// Employee Management System Class
class EmployeeManagementSystem {
private:
    vector<Employee*> employees;

public:
    void addEmployee(Employee* emp) {
        employees.push_back(emp);
    }

    void displayAllEmployees() {
        cout << "\nAll Employee Records:" << endl;
        for (Employee* emp : employees) {
            emp->displayDetails();
        }
    }

    void searchEmployeeByID(int id) {
        for (Employee* emp : employees) {
            if (emp->getEmployeeID() == id) {
                cout << "\nEmployee Found:" << endl;
                emp->displayDetails();
```

```
            return;
        }
    }
    cout << "\nEmployee with ID " << id << " not found." << endl;
}

~EmployeeManagementSystem() {
    for (Employee* emp : employees)
        delete emp;
}
};

// Main Function
int main() {
    EmployeeManagementSystem ems;

    Manager* mgr = new Manager();
    cout << "\nEnter Manager Details:" << endl;
    mgr->inputDetails();
    ems.addEmployee(mgr);

    Engineer* eng = new Engineer();
    cout << "\nEnter Engineer Details:" << endl;
    eng->inputDetails();
    ems.addEmployee(eng);
```

```
    ems.displayAllEmployees();


    int searchID;
    cout << "\nEnter Employee ID to search: ";
    cin >> searchID;
    ems.searchEmployeeByID(searchID);


    return 0;
}
```

4. Object-oriented programming enhances **modularity, reusability, scalability, and maintainability** in software development. **Encapsulation** protects data, **inheritance** promotes code reuse, **polymorphism** enables flexibility, and **abstraction** simplifies complexity. OOP leads to **efficient, secure, and versatile** applications.

b. **Use Abstract Classes**: Create a base Employee class with common methods, forcing derived classes like Manager and Engineer to have their own specific implementations. **Design Patterns**: Implement a Factory Pattern to create employee objects dynamically, allowing easier expansion of new employee types.

Data Retention: Add file handling or database support to save employee records permanently.

Error Handling: Include exception handling to manage invalid input or missing data smoothly.