Case Scenario: XYZ Software Solutions is developing an employee management system using C++. The system should manage employee records, including their names, IDs, and salaries. The company wants to implement Object-Oriented Programming principles such as encapsulation, inheritance, and polymorphism to make the system more efficient and maintainable. As a software developer, you are required to design and implement the core functionalities of the employee management system using OOP principles in C++. Assignment Tasks:

1. Introduction (5 Marks)

(a) Define Object-Oriented Programming and its significance.

(b)Explain the key OOP principles (Encapsulation, Inheritance, Polymorphism, and Abstraction) with examples.

Analysis of the Case Scenario (5 Marks)

1) Identify the key functional requirements of the employee management system.

2) Discuss how OOP principles can be applied to design the system effectively.

1. Introduction (5 Marks)

Object-Oriented Programming (OOP) and Its Significance

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "objects," which are instances of classes. OOP emphasizes the organization of software design around data, or objects, rather than functions and logic. Each object is an instance of a class, which is a blueprint that defines the structure and behaviors of the object.

The significance of OOP lies in its ability to model real-world entities, making it easier to design, understand, and maintain complex software systems. OOP helps in improving code reusability, scalability, and maintainability through its modular approach. It also facilitates the management of large codebases by enabling developers to create organized, easily understandable code that can be extended and modified efficiently.

Key OOP Principles

Encapsulation: Encapsulation is the concept of bundling the data (attributes) and methods (functions) that operate on the data into a single unit, or class. It restricts direct access to some of the object's components, making the object’s internal state protected from outside interference and misuse.

Example: In the context of an employee management system, an employee's salary should not be directly modified by external functions. Instead, it can be accessed or updated through well-defined getter and setter methods, ensuring validation or processing before any changes.

```
class Employee {
private:
double salary;
public:
void setSalary(double s) {
if (s >= 0) {
salary = s; } }
double getSalary() {
return salary; }
};
```

Inheritance: Inheritance allows one class (called a subclass or derived class) to inherit properties and behaviors (methods) from another class (called a superclass or base class). This promotes code reusability and establishes a hierarchy between classes.

Example: Suppose you have a Manager class that is derived from a general Employee class. A Manager inherits all properties of Employee, but it can also have additional properties such as a team size or department.

```
class Manager : public Employee {
private:
int teamSize;
public:
void setTeamSize(int size) {
teamSize = size; }
int getTeamSize() {
return teamSize; } };
```

Polymorphism: Polymorphism allows a single interface to be used for different data types or classes. It enables methods to be used in different ways, such as method overriding (in the case of inherited classes) and method overloading (same method name with different parameters).

Example: In an employee management system, a polymorphic method like calculateSalary() can be used for both Employee and Manager classes, but the implementation might differ for managers due to additional allowances or bonuses.

```
class Employee {
public:
virtual double calculateSalary() {
return 50000; // base salary for employee } }; class Manager : public Employee {
public: double calculateSalary() override {
return 70000; // base salary for manager } };
```

Abstraction: Abstraction is the concept of hiding the implementation details of a class and exposing only the essential features to the user. It simplifies complex systems by focusing on high-level functionalities and allowing access to only relevant data.

Example: The employee class may expose only essential details like employee ID, name, and salary without exposing internal details like tax calculations or salary breakdowns.

```
class Employee { private: int employeeID;
std::string name;
public:
Employee(int id, std::string name) : employeeID(id), name(name) {
void displayEmployeeDetails() {
std::cout << "Employee ID: " << employeeID << ", Name: " << name << std::endl; } };
```

2. Analysis of the Case Scenario (5 Marks)

Key Functional Requirements of the Employee Management System

The primary functional requirements of the employee management system include:

Employee Management:

Storing employee details like name, ID, salary, and other personal data.

Retrieving employee information based on ID or other attributes.

Updating employee data (e.g., salary changes).

Deleting employee records.

Role-based Management:

Employees may have different roles like "Manager," "Developer," etc. The system should support different functionalities based on roles.

Salary Management:

Calculating the salary of employees, with possible differences in pay based on role (e.g., manager vs developer).

Reports and Analytics:

Generating reports such as employee lists, salary reports, or performance evaluations.

Applying OOP Principles to Design the System Effectively

To design the employee management system efficiently, the following OOP principles can be applied:

Encapsulation:

Each employee will be encapsulated in a class with private data members such as employeeID, name, and salary. Methods to retrieve or modify these attributes will be provided via getter and setter functions.

This prevents unauthorized access or manipulation of the internal state of an employee object, ensuring better data integrity.

Inheritance:

Create a base Employee class with common attributes and methods, and then extend it for specific employee types such as Manager or Developer. This allows shared functionality to be reused while providing flexibility for specific roles.

For example, both Manager and Developer can inherit from the Employee class but override or extend functionality like salary calculation.

Polymorphism:

Use polymorphism to allow the system to treat different types of employees (e.g., Manager, Developer) uniformly while allowing for role-specific implementations of methods like calculateSalary(). This can be done by defining a virtual method in the base class and overriding it in derived classes.

Abstraction:

Expose high-level methods like addEmployee(), removeEmployee(), or generateReport() without requiring the user to understand the complex internal workings of employee management. This simplifies interactions with the system while hiding unnecessary details.

By following these principles, the employee management system can be designed to be modular, flexible, and maintainable, allowing future extensions (e.g., adding new employee roles or features) with minimal impact on existing code.