PART A: Conceptual Questions

1. Definition

In object-oriented programming, inheritance is the process of creating a new class(referred to as a child or subclass) from an existing class (referred to as parent or superclass). The child class can utilize the parent's methods and attributes as if they were its own since it inherits them. Additionally, it can override (customize) the parent class's existing behavior or add new features. Inheritance, composition, and aggregation are all approaches used to build complex objects, but they differ in structure and flexibility. Inheritance defines a "is-a" connection, in which a subclass extends a superclass to produce a more specialized version—for example, a "Dog is an Animal". While it encourages code reuse and hierarchy, it can become stiff or unnecessarily complex if not used properly. Composition, on the other hand, specifies a "has-a" connection in which one class includes another as part of its attributes—for example, a Car has an Engine. It's more versatile than inheritance and supports modular design. Aggregation is similar to composition, but with a weaker link; the contained object can exist independently of the container—for example, a Team contains Players, but the Players can exist without the Team.

2. Types

- Single Inheritance: a class inherits from one parent class. It is used when the subclass needs to specialize or extend the behavior of a single base class. Example:

```
class Animal:
    def eat(self):
        print("This animal eats food.")

class Dog(Animal):  # Single inheritance
    def bark(self):
        print("The dog barks.")
```

- Multiple Inheritance: A class inherits from two or more parent classes and, it's used when a class needs to combine features or behaviors from multiple unrelated classes. Example:

```
class Flyer:
    def fly(self):
        print("Flying high.")

class Swimmer:
    def swim(self):
        print("Swimming fast.")

class Duck(Flyer, Swimmer):  # Multiple inheritance
    pass
```

3. Overriding methods

Method overriding allows a derived class to redefine a method inherited from its base class while maintaining the same method name and signature. This allows the child class to customize or enhance the functionality to better meet its own requirements. For example, a

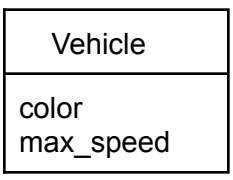
base class Animal may contain a generic speak() method, which a subclass Dog might override to offer more specialized behavior, such as barking. Overriding promotes polymorphism, letting objects behave differently depending on their class, increasing code flexibility and reusability. When you wish to change or personalize the behavior of a method that is already defined in the base class, you may override it rather than adding a new one in a derived class. Overriding ensures that the derived class can be used interchangeably with the base class, particularly in polymorphic situations—for example, when working with a list of base class objects that includes instances of derived classes. By overriding, the right version of the method is called based on the actual object type, allowing you to keep the interface constant while providing more precise functionality. Simply adding a new method wouldn't replace or modify the inherited behavior, and might lead to confusion or inconsistency when the base class method is still called in broader contexts.

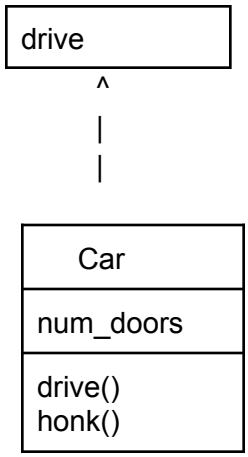
4. Real-World analogy

A real-life example of inheritance is the relationship between a driver's licence and a national ID card. A national ID card contains essential identity information, such as your name, date of birth, and citizenship. A driver's license retains these essential qualities (your name, date of birth, and so on) while adding particular features such as your license number, driving class, and expiry date. In this case, the driver's license "is a" sort of identification, similar to the ID card, but it is more precise and effective for a single purpose—driving. Just as in programming, the derived item (driver's license) expands on the basic concept (ID card) while keeping its key characteristics.

The driver's license example is consistent with the OOP idea of inheritance since it represents a "is-a" relationship—a driver's license is a form of ID. In programming, a subclass inherits the base class's essential properties (e.g., name and date of birth from the national ID) and adds or customizes features (e.g., license number, driving privileges). This is similar to how a subclass extends a superclass, reusing shared traits but adding more specialized behaviors or data.

PART B: UML DIAGRAM





The Car class inherits all basic features of Vehicle, including attributes like color and max_speed, as well as the general drive() method, through a "is-a" relationship with its parent. It then builds on that basis by adding its own property, num_doors, to monitor how many doors it has and introducing a new function, honk(), for sounding the horn. “Car” can override drive() to add gear-shifting logic or safety checks, while maintaining the essential driving behavior defined in Vehicle.

PART C: Short reflection & Discussion

1. When to use inheritance

In a graphical user interface library, a base Widget class provides basic rendering and event-handling methods. Subclasses such as Button, TextField, and Checkbox can inherit this behavior and add their own look and interaction logic. This eliminates duplicating boilerplate code and makes it simple to treat all widgets consistently (for example, placing them in layouts or handling focus).

Assume you develop a long inheritance chain to combine unrelated functionality, such as a class AudioLoggerTimerCache that derives from AudioProcessor, Logger, Timer, and CacheManager. Multiple inheritance can cause "diamond problem," tight coupling, and unanticipated side effects when methods overlap, making the design brittle. Composition (keeping distinct helper objects) provides a cleaner and more maintainable solution.

2. Method overriding vs. overloading

Method overriding (runtime polymorphism) occurs when a subclass provides a new implementation of a method inherited from a parent class, with the same name and parameters. It is resolved at runtime, allowing for dynamic behavior depending on the object type.

Method overloading (compile-time polymorphism) occurs when many methods in the same class have the same name but differ in parameter lists. It is resolved during compile time.

Overriding is used in inheritance to allow for flexibility and dynamic behavior—subclasses can tweak or extend parent methods, allowing different objects to respond individually to the same method call, which is essential for polymorphism and reusable, adaptable code.

3. Inheritance vs. interfaces/abstract classes

Inheritance occurs when a class extends another class, inheriting its implementation and structure, but implementing an interface (or abstract base class) indicates that a class agrees to

provide specified functions without inheriting actual code. Abstract classes can provide partial implementation, although interfaces normally simply specify method signatures. Inheritance allows you to reuse code, whereas interfaces create contracts for consistent behavior across distinct classes, allowing you to be flexible and implement multiple ways.

4. Pitfalls of Multiple Inheritance

The diamond problem is a potential issue with multiple inheritance, in which a class inherits from two classes that both inherit from the same base class, causing ambiguity regarding which attributes or methods to utilize.
A frequent technique for avoiding this issue is to use interface-based design (common in Java and C#), in which classes implement numerous interfaces rather than inheriting behavior. In C++, virtual inheritance ensures that the base class is only inherited once, reducing duplication and ambiguity.

PART D: Research(Optional)

1. Inheritance in different Languages

Inheritance in Python vs. C++
Python

- Python uses the **Method Resolution Order (MRO)** to handle multiple inheritance and resolve ambiguity.
- Classes are created using the `class` keyword.
- It supports **dynamic typing**, so type checking is done at runtime.

C++

- **Supports single and multiple inheritance**, but more complex due to strict type checking.
- Multiple inheritance can lead to the **diamond problem**, which can be resolved using **virtual inheritance**.
- It is **statically typed**, so all types must be known at compile time.

Restrictions

- **Python** has more flexibility but less compile-time safety.
- **C++** provides stricter access control and compile-time checking, but multiple inheritance is more prone to complexity and errors if not handled carefully.

2. Open-Closed Principle

The **Open-Closed Principle** from SOLID states that *software entities (classes, modules, functions, etc.) should be open for extension but closed for modification*. This means you should be able to add new functionality by extending existing code—**not by changing it**.

Inheritance supports this principle by allowing a **derived class** to add or override functionality without altering the base class itself. This promotes stability and reusability, especially when the base class is part of a tested and deployed library.

Example:

```cpp
#include <iostream>
using namespace std;

// Base class
class Notification {
public:
    virtual void send(const string& message) {
        cout << "Sending generic notification: " << message << endl;
    }

    virtual ~Notification() {}  // Virtual destructor for safe polymorphic deletion
};

// Derived class 1
class EmailNotification : public Notification {
public:
    void send(const string& message) override {
        cout << "Sending email: " << message << endl;
    }
};

// Derived class 2
class SMSNotification : public Notification {
public:
    void send(const string& message) override {
        cout << "Sending SMS: " << message << endl;
    }
};

// Client code using polymorphism
void notifyUser(Notification* notifier, const string& message) {
    notifier->send(message);
}

int main() {
    EmailNotification email;
    SMSNotification sms;

    notifyUser(&email, "Welcome to the system!");
    notifyUser(&sms, "Your verification code is 123456.");

    return 0;
}
```