OOPS NOTES

What is object oriented programming ?

Object-Oriented Programming (OOP) is a programming paradigm or approach that organises and structures code around the concept of "objects." An object is a self-contained unit that combines data (attributes) and functions (methods) that operate on that data. This approach allows developers to model real-world entities or concepts in their code and provides a way to manage complexity and promote reusability.

Key concepts of Object-Oriented Programming include:

1. Classes and Objects: A class is a blueprint or template that defines the structure and behaviour of objects. It encapsulates data (attributes) and methods (functions) that operate on that data. An object is an instance of a class, created based on the class's blueprint.
2. Encapsulation: This refers to the bundling of data and methods that operate on that data within a single unit (object or class). It helps in hiding the internal implementation details of an object, exposing only the necessary interfaces for interacting with it. Encapsulation helps maintain code organisation and improves data security.
3. Abstraction: Abstraction involves focusing on essential properties and behaviours of an object while ignoring the unnecessary details. It allows developers to create models that simplify complex real-world systems and focus on what's relevant to the problem at hand.
4. Inheritance: Inheritance enables the creation of new classes (derived or subclass) that inherit attributes and methods from an existing class (base or superclass). This promotes code reuse and the establishment of hierarchical relationships between classes.
5. Polymorphism: Polymorphism allows objects of different classes to be treated as objects of a common superclass. This enables more flexible and generic programming by allowing different classes to be used interchangeably through a shared interface.

Benefits of Object-Oriented Programming include:

1. Modularity: Code is organised into self-contained objects, making it easier to manage and understand. Changes to one part of the code have less impact on other parts.
2. Reusability: Objects and classes can be reused in different parts of the program or in different programs altogether, reducing development time and effort.
3. Maintainability: OOP's modular nature and encapsulation make it easier to maintain and debug code. Changes to one object's implementation do not necessarily affect other objects.
4. Flexibility: OOP allows for the creation of abstract classes and interfaces, enabling developers to build systems that can evolve and adapt to changing requirements.
5. Scalability: Object-oriented systems can be scaled more easily by extending existing classes or creating new classes, which helps in handling complex projects.

How object oriented programming is related to the real world?

Object-Oriented Programming (OOP) is closely related to the real world because it is designed to model the way we conceptualise and interact with objects and concepts in our everyday lives. OOP provides a programming paradigm that mirrors how we think about and categorise objects, their properties, and their behaviours . Let's explore this relationship with a real-world example:

Example: Modelling a Car

Imagine you're tasked with creating a program that simulates a car and its interactions. You can use OOP principles to model the car as an object, complete with its attributes (data) and behaviours (methods). Here's how OOP relates to the real world in this example:

1.Class and Object Representation: In the real world, a "car" is a tangible object with certain attributes (colour , make, model) and behaviours (start, accelerate, brake). In OOP, you'd create a class named Car that serves as a blueprint for creating individual car objects. Each car object you create from this class will have its own unique attributes and behaviours .
2.Encapsulation: A car has internal mechanisms that the driver doesn't need to understand fully. Similarly, in OOP, you can encapsulate the internal details of a car's implementation within its class. This means you can hide complex implementation details and expose only the methods that allow external interactions (e.g., starting the engine).
3.Abstraction: When you drive a car, you don't need to know every intricate detail of how the engine works. You interact with a simplified interface—steering wheel, pedals, dashboard—that abstracts away the complexity. In OOP, you can define an abstract class Vehicle with common attributes and methods that multiple vehicle types (cars, trucks, motorcycles) can inherit from.
4.Inheritance:Different car models might share common features (e.g., all cars have wheels and an engine), but they can also have unique characteristics. OOP's inheritance allows you to create a base class (Car) with common attributes and behaviours and then derive specific car models (subclasses) from it. This promotes code reuse and maintains a clear hierarchical relationship.

What are the limitations of oops ?

1.Complexity: OOP can sometimes lead to complex class hierarchies and relationships, which might make the code harder to understand and maintain. Overuse of inheritance or poorly designed class structures can result in confusion and increased cognitive load.
2.Performance Overhead: OOP can introduce performance overhead due to the dynamic nature of method calls, object creation, and memory management. Virtual method calls and additional layers of abstraction can impact execution speed, especially in performance-critical applications.
3.Overhead of Abstraction: While abstraction is a strength of OOP, excessive abstraction can lead to code that is disconnected from the underlying logic, making it difficult to understand and debug. Balancing the right level of abstraction is crucial.
4.Limited Support for Parallel Programming: OOP might not be well-suited for parallel programming, where tasks need to be divided and executed concurrently. Managing shared state and synchronisation can be challenging within an OOP framework.
5.Lack of Direct Access to Data: Encapsulation, a key principle of OOP, restricts direct access to an object's internal data. While this enhances data security and modularity, it can lead to additional layers of method calls when simple data access is needed.
6.Inheritance Hierarchy Issues: Over-reliance on inheritance can lead to inflexible and tightly coupled code. Changes in the base class can have unintended effects on derived classes, making the codebase fragile.

Aspect	Class	Object
Definition	A class is a blueprint or template that defines the structure and behavior of objects. It's a template for creating objects.	An object is an instance of a class. It is a concrete representation of a specific entity based on the class's blueprint.
Usage	Classes serve as a blueprint for creating multiple objects of the same type.	Objects are individual instances that hold specific data and state.
Attributes	Classes define attributes (data members) that are shared by all objects of that class.	Objects have attributes that store specific values for that instance.
Methods	Classes define methods (functions) that operate on the attributes and perform actions related to the class's concept.	Objects can call methods defined in the class to perform actions or operations specific to that instance.
Creation	Classes are used as a blueprint to create objects.	Objects are created using the class's constructor method.
Memory Allocation	Memory is allocated for a class's attributes and methods in the program's memory space.	Memory is allocated for each object instance, which includes its attributes and a reference to the class's methods.
Examples	If "Car" is a class, it might define attributes like "make" and "model," as well as methods like "start" and "stop."	An object of the class "Car" could be a specific car instance with attributes like "Toyota" and "Camry."

Static keyword and why it is used ?

The static keyword is used in programming languages to define class-level members that belong to the class itself rather than to individual instances (objects) of the class. This keyword has different implications based on the context in which it is used. In many programming languages, such as Java, C++, and C#, the static keyword can be applied to variables, methods, and nested classes.

Static Variables:
When a variable is declared as static within a class, it becomes a class-level variable rather than an instance-level variable. This means that all instances of the class share the same variable, and changes made to it are reflected across all instances.

Static Methods:
A static method is a method that belongs to the class itself, not to any specific instance. It can be called using the class name, without needing to create an object of the class. These methods are typically used for utility functions or operations that don't require access to instance-specific data.

Advantages of Using static:
1.Memory Efficiency: Static members are shared among all instances of the class, leading to memory savings when the same data is needed across instances.
2.Global Access: Static methods and variables can be accessed directly using the class name, making them accessible from anywhere in the code without the need for an instance.
3.Utility Functions: Static methods are useful for defining utility functions that don't require access to instance-specific data.
4.Constants: static can be used to declare constants that are relevant to the class as a whole.
5.Performance: Static members often offer better performance due to reduced memory allocation and method call overhead.

Virtual Keyword and why it is used ?

The virtual keyword is used in object-oriented programming languages, such as C++ and C#, to indicate that a method in a base class is meant to be overridden (redefined) by derived classes. When a method is marked as virtual in a base class, it allows derived classes to provide their own implementations of that method, enabling polymorphism and dynamic method dispatch.

Defining Virtual Methods:
When a method is declared as virtual in a base class, it signals that the method's behaviour can be customised in derived classes. This is particularly useful when you want to provide a default implementation in the base class while allowing subclasses to modify or extend that behaviour .

Advantages of Using virtual:
1.Polymorphism: virtual methods enable you to work with objects of different derived classes through a common interface, allowing for more flexible and generic code.
2.Behaviour Customisation : virtual methods allow derived classes to customise the behaviour of methods inherited from the base class, tailoring them to their specific needs.
3.Extensibility: Base classes can provide a default implementation that can be reused across derived classes, while still allowing those derived classes to extend or modify the behaviour .
4.Code Organisation : virtual methods contribute to a well-structured and organised codebase by facilitating clear separation between common behaviour in the base class and specific behaviour in derived classes.

Abstract Keyword and why it is used ?

The abstract keyword is used in object-oriented programming languages, such as Java, C#, and C++, to declare a class or a method as incomplete or lacking a full implementation. An abstract class cannot be instantiated directly, and an abstract method must be overridden by any non-abstract subclass. The abstract keyword is used to create a foundation for building classes with common attributes and methods while ensuring that certain behaviours are implemented in derived classes.

Abstract Classes:
An abstract class is a class that cannot be instantiated on its own and is meant to serve as a blueprint for other classes. It may contain a mix of concrete methods (methods with implementation) and abstract methods (methods without implementation). An abstract class can have instance variables, constructors, and concrete methods that can be inherited by subclasses.

Advantages of Using abstract:
1.Enforcing Implementation: Abstract classes and methods provide a way to define a common interface for derived classes while ensuring that certain methods must be implemented in subclasses.
2.Polymorphism: Abstract classes enable polymorphism by allowing you to work with objects of different concrete subclasses through a common abstract class reference.
3.Code Reusability: Abstract classes can provide shared methods and attributes that are inherited by multiple subclasses, promoting code reusability and maintainability.
4.Consistent Interface: Abstract classes define a consistent interface that derived classes adhere to, making it easier to understand and work with different classes that share a common purpose.
5.Customisation : Abstract methods let derived classes customise behaviour specific to their implementation while adhering to the overall structure defined in the abstract class.

Final Keyword and why it is used ?

The final keyword is used in programming languages like Java, C++, and C# to indicate that a class, method, variable, or parameter cannot be further modified or extended by subclasses, overridden by derived methods, or reassigned new values. It serves as a modifier that provides a level of restriction and control over various elements in a program.

Advantages of Using final:
1.Preventing Modifications: The primary benefit of using final is to ensure that certain elements in your code remain constant and cannot be accidentally modified, leading to safer and more predictable behaviour .
2.Security and Consistency: final helps maintain the integrity of code by preventing unintended changes, reducing the risk of introducing bugs and security vulnerabilities.
3.Method Safety: Marking methods as final ensures that the behaviour of these methods cannot be altered in subclasses, providing a clear contract for derived classes to follow.
4.Code Optimisation : In some cases, the use of final can provide hints to compilers, enabling them to perform certain optimisations that might not be possible with mutable elements.
5.API Design: Using final can help define a stable and reliable interface for libraries or APIs, as it indicates which elements are not meant to be extended or overridden by external users.

what is explicit keyword and why it is used ?

The explicit keyword is used in C++ and some other programming languages to control the automatic type conversion that occurs during object construction. It is particularly relevant when dealing with constructors that can be implicitly invoked by the compiler for certain conversions. By using the explicit keyword, you can prevent unintended and potentially ambiguous type conversions that might lead to unexpected behaviour .

Advantages of Using explicit:
1.Preventing Ambiguity: The primary benefit of using explicit is to prevent unintended and ambiguous type conversions that might lead to unexpected behaviour and potential bugs.
2.Clarity and Control: Marking constructors as explicit makes the code more readable by making it clear when type conversions are being intentionally performed.
3.Safer Code: By using explicit, you minimise the chances of subtle and implicit conversions that can lead to hard-to-debug errors.

This keyword and why it is used ?

The this keyword is a reference to the current instance of a class in object-oriented programming languages like Java, C++, and C#. It is used within the context of an instance method or constructor to refer to the object on which the method is invoked or the constructor is being called. The this keyword is especially useful in scenarios where there might be ambiguity between instance variables and method parameters with the same name.

Advantages of Using this:
1.Clarity: The this keyword makes your code more readable by explicitly indicating when you're referring to instance variables or methods within the current instance.
2.Preventing Errors: Using this helps avoid mistakes and confusion that might arise from having variables with the same name or from mistakenly using local variables instead of instance variables.
3.Method and Constructor Chaining: The this keyword facilitates chaining methods or constructors, leading to cleaner and more concise code.
4.Consistency: The this keyword promotes consistency in your codebase, ensuring that you're accessing instance-level elements when needed.
5.Explicitness: By using this, you clearly indicate your intention to refer to the current instance, improving the understanding of your code's behaviour .

New keyword and why it is used ?

The new keyword is used in object-oriented programming languages to dynamically allocate memory for objects at runtime and create instances of classes. It is used to instantiate (create) objects and invoke constructors to initialise the newly allocated memory. The new keyword is an essential part of creating objects and managing memory in languages like Java, C++, C#, and many others.

Advantages of Using new:
1.Dynamic Memory Allocation: The new keyword allows you to allocate memory for objects dynamically at runtime. This enables you to create objects as needed and release memory when it's no longer required.
2.Object Creation: Using new is the primary way to create instances of classes, allowing you to work with objects that represent real-world entities or concepts.

Explain const ?

The const keyword is used in programming languages to indicate that a variable, function parameter, or method cannot be modified or altered after its initial assignment. It's used to define constants, enforce immutability, and improve code reliability by preventing accidental changes to values that are meant to remain constant.

Advantages of Using const:
1.Code Reliability: The primary benefit of using const is that it helps prevent accidental modifications to variables, parameters, and methods that are meant to remain constant or unchanged.
2.Readability: Marking variables and parameters as const improves code readability by clearly indicating which values should not be modified.
3.Enhanced Debugging: const can help catch unintended changes early in development, making debugging easier and reducing the likelihood of hard-to-find bugs.
4.Safer APIs: const parameters and methods provide safer APIs by ensuring that certain operations won't modify the provided data or object's state.
5.Enforcing Design Contracts: const enforces design contracts and expectations by clearly specifying which elements are intended to be immutable.

Explain super keyword ?

The super keyword is used in object-oriented programming languages like Java, Python, and some others to refer to the superclass (parent class) of the current subclass (child class). It is primarily used to access members (fields or methods) of the superclass from the subclass, allowing for method overriding, constructor chaining, and accessing overridden methods or fields.

Advantages of Using super:
1.Method Overriding and Extension: The primary benefit of using super is to facilitate method overriding while still being able to access and extend the behaviour of the superclass's version of the method.
2. Code Reuse: super helps in reusing initialisation logic or behaviour from the superclass in the subclass, promoting code reuse and maintaining a consistent behaviour across related classes.
3. Constructor Chaining: In languages that support constructor chaining, the super keyword lets you invoke constructors in the superclass to perform common initialisation tasks.
4. Polymorphism: Using super allows you to work with objects of different subclasses through a common superclass reference, contributing to polymorphic behaviour .

What is polymorphism and why do we need it ?

Polymorphism is a fundamental concept in object-oriented programming that allows objects of different classes to be treated as instances of a common base class. It enables you to write more generic, flexible, and reusable code by providing a way to work with different types of objects through a common interface. Polymorphism allows objects to exhibit different behaviors based on their actual types, leading to code that's easier to maintain and extend.
why we need it:
1.Common Interface: Polymorphism allows you to define a common interface, typically through a base class or interface, that multiple classes can implement. This common interface defines a set of methods that all derived classes must provide, ensuring a consistent way to interact with different types of objects.
2.Method Overriding: Polymorphism is closely related to method overriding. When a derived class implements a method declared in its base class, it's said to override that method. This allows each subclass to provide its own specialised implementation while adhering to the interface defined in the base class.
3.Code Reusability: Polymorphism promotes code reusability by allowing you to write code that can work with a wide range of different objects that share a common interface. This reduces the need for duplicated code or writing separate code for each specific class.

Explain compile time polymorphism , and what function cannot be overloaded in c++ ?

Compile-time polymorphism, also known as static polymorphism or method overloading, occurs when the appropriate method or function to call is determined by the compiler at compile time based on the number and types of arguments provided. It involves having multiple methods or functions with the same name but different parameter lists. The compiler selects the appropriate version of the method to call based on the arguments passed to it.

cpp Copy code

```cpp
class Calculator {
public:
    int add(int a, int b) {
        return a + b;
    }

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

int main() {
    Calculator calc;
    int result1 = calc.add(2, 3);        // Calls the int version
    double result2 = calc.add(2.5, 3.7); // Calls the double version
    return 0;
}
```

In C++, some functions cannot be overloaded due to ambiguity or special behavior. These include:

1. Function Overloading and Default Arguments: You cannot have two overloaded functions where one uses default arguments and the other doesn't. This can lead to ambiguity when the function is called with no arguments.
2. Operator Overloading: Certain operators have fixed meanings and cannot be overloaded with different meanings. For example, the . (dot) operator for member access and the :: operator for scope resolution cannot be overloaded.
3. Overloading Based Only on Return Type: Overloading based solely on the return type of a function is not allowed, as it can lead to ambiguity since function calls are determined based on the arguments, not just the return type.

Here are the operators that cannot be overloaded in C++:

1. Scope Resolution Operator (::): The scope resolution operator is used to specify the namespace or class scope of a name. It cannot be overloaded, as it's necessary for specifying the scope of functions, variables, and classes.
2. Member Access Operators (. and ->): The member access operators (. and ->) are used to access members of classes and structures. These operators cannot be overloaded because they are fundamental to the syntax and semantics of the language.
3. Conditional Operator (?:): The conditional operator (ternary operator) is used for conditional expressions. Overloading it could lead to ambiguity in expressions and make the code more complex and difficult to understand.

Explain function overriding ?

Function overriding is a concept in object-oriented programming where a subclass provides a specific implementation for a method that is already defined in its superclass. In other words, when a subclass has a method with the same name, return type, and parameter list as a method in its superclass, it is said to be overriding the superclass method. This allows the subclass to customise or extend the behaviour of the inherited method.

Here's an explanation of function overriding and how it works:

1. Inheritance and Base Methods: In an inheritance hierarchy, a subclass inherits methods and attributes from its superclass. If a subclass wants to modify the behaviour of a method inherited from its superclass, it can override that method by providing its own implementation.
2. Method Signature: For a method to be considered overridden, it must have the same method signature in both the superclass and the subclass. This includes the method's name, return type, and parameter list.

Advantages of Function Overriding:

1. Customised Behaviour : Overriding allows a subclass to provide a specialised implementation of a method, tailoring it to its specific needs.
2. Polymorphism: Function overriding contributes to polymorphic behaviour , where an object can be treated as an instance of its superclass or subclass interchangeably.
3. Code Reuse: Overriding allows the reuse of method names and signatures, providing a consistent interface while allowing for class-specific behaviour .

Explain Run time polymorphism ?

Runtime polymorphism, also known as dynamic polymorphism, is a key concept in object-oriented programming that allows different classes to be treated as instances of a common base class through a shared interface. It enables the selection of the appropriate method implementation at runtime based on the actual type of the object, rather than the reference type. This mechanism allows for more flexible and dynamic behaviour in object-oriented systems.

Advantages of Runtime Polymorphism:

1.Flexibility and Extensibility: Runtime polymorphism allows new subclasses to be added to an existing hierarchy without modifying existing code. This promotes code extensibility and adaptability.
2.Polymorphic Behaviour : Objects of different subclasses can be treated interchangeably, promoting polymorphic behaviour . This simplifies code and improves maintainability.
3.Single Interface: Runtime polymorphism enforces a common interface for different objects, enabling a clear separation between interface and implementation.

A virtual function is a concept in object-oriented programming that allows a method in a base class to be overridden by a method in a derived class. It enables runtime polymorphism, allowing objects of different derived classes to be treated as instances of a common base class and to invoke the appropriate method implementation based on the actual type of the object. Virtual function can be set private.

What is a virtual class ?

Virtual Base Class:
One common interpretation of a "virtual class" could refer to a base class that is marked as "virtual." In multiple inheritance scenarios, the use of a virtual base class ensures that there's only one instance of that base class shared among all the derived classes. This avoids the "diamond problem" where ambiguity arises due to multiple paths to a common base class.

Abstract Class:
Another interpretation could be an abstract class, which is a class that cannot be instantiated and is meant to serve as a base for other classes. An abstract class may have virtual functions that are intended to be overridden by its derived classes, enabling dynamic polymorphism.

What is Derived class ?

A derived class, also known as a subclass or child class, is a class that is derived from another class, called the base class or parent class. In object-oriented programming, a derived class inherits attributes and methods from its base class and can also define its own attributes and methods. The concept of inheritance allows for the creation of class hierarchies, where more specialised classes inherit properties and behaviour from more general classes.

What is pure virtual function and a pure virtual destructor ?

A pure virtual function, also known as an abstract method, is a virtual function declared in an abstract class that has no implementation in the base class. This function is meant to be overridden by concrete (non-abstract) subclasses to provide specific implementations. The existence of pure virtual functions in an abstract class makes that class itself abstract, meaning it cannot be directly instantiated.

A pure virtual destructor is a special type of destructor in C++ that is declared as pure virtual in a base class. Just like pure virtual functions, a pure virtual destructor has no implementation in the base class and is intended to be overridden by derived classes. The main purpose of a pure virtual destructor is to ensure that derived classes correctly release resources when they are destroyed, even when polymorphism is involved.

what is encapsulation and the concept of data hiding and abstraction explain?

Encapsulation, data hiding, and abstraction are fundamental concepts in object-oriented programming (OOP) that help organise and structure code, improve code reusability, and maintainability. They promote the creation of well-structured and modular software systems. Let's delve into each of these concepts:

1.Encapsulation: Encapsulation is the concept of bundling data (attributes) and the methods (functions) that operate on that data into a single unit, known as a class. It involves restricting direct access to an object's internal data and providing controlled access through methods. Encapsulation helps hide the implementation details and enforces a clear separation between an object's internal representation and the way it's used by the outside world.
2.Data Hiding: Data hiding, also known as information hiding, is a key aspect of encapsulation. It involves making an object's internal data private, so that it can only be accessed and modified through the methods provided by the class. By hiding the implementation details, you prevent unintended modifications or misuse of the data, ensuring that only valid and controlled interactions occur.
3.Abstraction: Abstraction is the process of simplifying complex reality by modelling classes based on real-world entities or concepts. It involves identifying the essential characteristics of an object while ignoring irrelevant details. Abstraction allows you to create a high-level view of an object's functionality, providing a clear and concise interface for interaction without exposing the internal complexities.

Needs of Encapsulation:

1.Data Protection: Encapsulation ensures that an object's internal data is protected from direct external access and modification. This prevents accidental or unauthorised changes that could lead to errors or data corruption.
2.Controlled Access: Encapsulation allows you to control how data is accessed and modified by providing well-defined methods. This prevents external code from bypassing the intended behaviour and rules.
3.Maintenance: When data and methods related to that data are encapsulated within a class, changes and updates can be made to the internal implementation without affecting the rest of the codebase. This makes maintenance and updates easier and less error-prone.

Advantages of Encapsulation:

1.Data Hiding: Encapsulation enables data hiding, which means that the internal details of an object are hidden from the outside world. Only the methods provided by the class can interact with the data, reducing the risk of misuse.
2.Modularity: Encapsulation promotes modularity by allowing you to divide a complex system into smaller, manageable components (classes). Each class encapsulates a specific set of functionality, making the codebase more organised and easier to understand.
3.Reusability: Encapsulated classes can be reused in different parts of an application or in different applications. Since the interface (methods) is standardised , you can use the same class in various contexts without worrying about the internal implementation.
4.Security: By controlling access to data and methods, encapsulation enhances the security of the codebase. Sensitive data can be hidden from external code, reducing the risk of unauthorised access.
5.Flexibility: Encapsulation provides flexibility by allowing you to modify the internal implementation of a class without affecting the code that uses the class. This is especially important when making changes to improve performance or add new features.

Access Modifiers
public: Members declared as public are accessible from any code that has access to an object of the class.
protected: Members declared as protected are accessible within the class and its derived classes.
private: Members declared as private are accessible only within the class itself.

Aspect	Encapsulation	Abstraction
Purpose	Data protection and controlled access.	Simplifying complex reality by modeling classes based on essential characteristics.
Focus	Protecting internal data and details.	Defining high-level interface and functionality while hiding implementation.
Implementation	Achieved through access specifiers.	Achieved through identifying essential attributes and methods.
Visibility	Public, protected, and private access levels.	Public interface is visible; implementation details are hidden.
Members Visibility	Members can be variously accessible.	Members can be declared with different access levels.
Interaction	External code interacts through methods.	External code interacts with abstracted interface.
Example	Getter and setter methods for private attributes.	Interface of a class that focuses on essential methods and attributes.

Explain Inheritance ?

Inheritance is a fundamental concept in object-oriented programming (OOP) that allows you to create a new class (subclass or derived class) based on an existing class (superclass or base class). Inheritance promotes code reuse and the creation of class hierarchies, where more specialised classes inherit properties and behaviours from more general classes.

Here's an explanation of inheritance, subclasses, superclasses, and the concept of reusability:

Inheritance:
Inheritance is the process of creating a new class (subclass) that inherits attributes and methods from an existing class (superclass). The subclass can extend or override the behaviour of the superclass while inheriting its characteristics.

Superclass (Base Class):
A superclass, also known as a base class or parent class, is the existing class from which another class (subclass) is derived. It provides a common set of attributes and methods that can be shared among multiple subclasses.

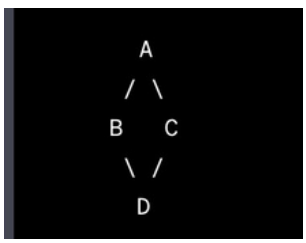
Subclass (Derived Class):
A subclass, also known as a derived class or child class, is a new class created by inheriting properties and behaviours from a superclass. The subclass can add new attributes and methods, and it can also override or extend the behaviour of inherited methods.

1.Single Inheritance: In single inheritance, a subclass inherits from a single superclass. This is the simplest form of inheritance, where each subclass has only one immediate superclass. It promotes a linear hierarchy of classes.
2.Multiple Inheritance: In multiple inheritance, a subclass can inherit from multiple superclasses. This allows a class to acquire properties and behaviours from multiple sources. However, multiple inheritance can lead to complexities, such as the diamond problem, which arises when a subclass inherits from two or more classes that have a common superclass.
3.Hierarchical Inheritance: In hierarchical inheritance, multiple subclasses inherit from a single superclass. This creates a tree-like structure where a superclass serves as the root, and multiple subclasses extend from it.
4.Multilevel Inheritance: In multilevel inheritance, a subclass inherits from another subclass. This creates a chain of inheritance, where each subclass inherits properties and behaviours from its parent class. It allows for extending classes in a layered manner.
5.Hybrid Inheritance: Hybrid inheritance is a combination of two or more types of inheritance, such as single, multiple, hierarchical, or multilevel. It allows for creating complex class relationships that can be tailored to specific design requirements.
6.Virtual Inheritance: Virtual inheritance is used in cases of diamond inheritance (a form of multiple inheritance) to prevent ambiguity and the diamond problem. It ensures that only one instance of a common base class is inherited by the derived class, even if that base class appears multiple times in the inheritance hierarchy.

•The sealed modifier is a feature found in some object-oriented programming languages, such as C# and Java, and it's used to restrict the inheritance of a class or the overriding of a method. When a class or method is marked as sealed, it cannot be further derived (in the case of classes) or overridden (in the case of methods) by any other class or subclass.

what is the diamond problem in case of multiple inheritance ?

The "diamond problem" is a term used in the context of multiple inheritance in object-oriented programming to describe a situation where a class inherits from two or more classes that have a common base class. This can lead to ambiguity and challenges when resolving which version of a method or attribute should be used in the derived class.



What is Object Slicing and Friend Function ?

Object slicing is a concept that occurs in object-oriented programming languages like C++ when a derived class object is assigned to a base class object, resulting in the loss of specialised properties and behaviours of the derived class.

Friend Function:
In C++, a friend function is a function that is not a member of a class but has access to the private and protected members of that class. It is declared using the friend keyword inside the class, and it allows the friend function to access private or protected members of the class, as if it were a member function.

Aspect	Static Memory Allocation	Dynamic Memory Allocation
Definition	Memory allocation for variables that have a fixed size and lifetime throughout the program's execution.	Memory allocation for variables whose size and lifetime are determined during runtime.
Storage Location	Stack (local variables, function parameters)	Heap (dynamically allocated memory)
Allocation Time	Compile-time	Runtime
Deallocation	Automatically released when the variable goes out of scope (e.g., end of function)	Requires explicit deallocation to avoid memory leaks.
Memory Management	Managed by the compiler	Managed by the programmer
Flexibility	Less flexible since memory requirements are known in advance.	More flexible since memory can be allocated and resized as needed.
Scope	Limited to the scope in which the variable is defined.	Can be passed across different parts of the program.
Examples	int x;	int* ptr = new int;

Aspect	Pointers	References
Definition	A variable that holds the memory address of another variable.	An alias or alternative name for an existing variable.
Declaration Syntax	Type* variableName;	Type& variableName;
Initialization	Can be uninitialized or assigned nullptr.	Must be initialized when declared.
Null Value	Can be assigned a nullptr to indicate no valid address.	Cannot directly hold null or nullptr value.
Memory Address	Holds the actual memory address of the pointed-to variable.	Acts as an alias; no memory address stored.
Indirection	Requires dereferencing using * operator to access the value pointed to.	Directly accesses the value through the reference.
Pointer Arithmetic	Supports arithmetic operations like addition and subtraction.	Doesn't support arithmetic operations.
Reassignment	Can be reassigned to point to different variables or locations.	Once initialized, cannot be made to reference a different variable.
Memory Management	Requires explicit memory deallocation using delete operator.	No need for explicit memory management.

What is garbage collection ?

Garbage collection is a memory management technique used in programming languages to automatically reclaim memory that is no longer needed by the program. It helps prevent memory leaks and frees programmers from the responsibility of explicitly managing memory deallocation. However, it's important to note that C++ does not have a built-in garbage collection mechanism like some other programming languages such as Java or C#. Instead, C++ relies on manual memory management using techniques like destructors and smart pointers.

What is exception ?

An exception in programming refers to an abnormal or unexpected event that occurs during the execution of a program, disrupting its normal flow. Exceptions are typically caused by errors or exceptional conditions that can arise during program execution. They allow the program to handle unexpected situations gracefully and provide a mechanism for propagating information about the error from where it occurred to where it can be handled.

Error:
An error generally refers to a deviation from the expected or desired behaviour of a program. Errors can encompass a wide range of issues, including syntax errors (e.g., typos in the code), logic errors (e.g., incorrect calculations), and runtime errors (e.g., division by zero, accessing an out-of-bounds array element). Errors can result in program crashes, incorrect output, or unexpected behaviour .

Exception:
An exception, on the other hand, specifically refers to an object or piece of data that is created and thrown during program execution to indicate that something exceptional or erroneous has occurred. Exceptions are part of a mechanism designed to handle errors in a structured way. They allow the program to continue running even when errors occur by providing a way to transfer control to an exception handler, where the error can be handled appropriately.

Exception handling in C++ involves using the try, catch, and throw keywords to manage and handle exceptions that occur during the execution of a program. Exception handling allows you to gracefully handle errors and exceptional conditions, preventing the program from crashing and providing a way to recover from unexpected situations.

try Block:
The try block is used to enclose the code that might raise an exception. It's where you specify the code that could potentially throw an exception.

catch Blocks:
catch blocks are used to handle specific types of exceptions that may be thrown within the corresponding try block. Each catch block specifies a particular exception type it can handle. If an exception of that type is thrown within the try block, the corresponding catch block is executed.

throw Statement:
The throw statement is used to explicitly throw an exception. It generates an exception object of a specified type and can include additional information about the error.

What is enum ?

In C++, an enumeration, often referred to as an "enum," is a user-defined type that consists of a set of named constant values. Enums provide a way to create symbolic names for integer values, making the code more readable and self-explanatory. Enums are commonly used to represent a finite set of related values, such as days of the week, months, error codes, and more.

Here's a short tabular comparison between run-time and compile-time polymorphism:

Aspect	Compile-time Polymorphism (Static Binding)	Run-time Polymorphism (Dynamic Binding)
Timing	Occurs at compile time.	Occurs at runtime.
Mechanism	Achieved through method overloading and operator overloading.	Achieved through method overriding.
Decision	Method resolution is determined by the compiler based on the method signature.	Method resolution is determined by the runtime environment based on the actual object type.
Examples	Function overloading in C++, method overloading in Java.	Method overriding in Java and C#.
Flexibility	Provides less flexibility as method calls are bound at compile time.	Provides more flexibility as method calls are bound at runtime, allowing for late binding and dynamic dispatch.
Performance	Generally faster due to static binding.	May incur a slight performance overhead due to dynamic method resolution.

- **Shallow Copy**: Shallow copy creates a new object and then copies the values of the attributes from the original object to the new object. However, if the attributes themselves are references to other objects, the shallow copy creates new references to the same objects instead of copying the referenced objects themselves. As a result, changes made to the referenced objects in the new object will affect the original object, and vice versa.
- **Deep Copy**: Deep copy creates a new object and then recursively copies the values of all attributes, including any referenced objects, from the original object to the new object. This ensures that each object in the new hierarchy is completely independent of the corresponding objects in the original hierarchy. Changes made to the referenced objects in the new object will not affect the original object, and vice versa. Deep copy is typically more resource-intensive than shallow copy, as it involves duplicating the entire object hierarchy.