**Assignment 3**

**Que 1. What is an object in C++?**

**Ans:** C++ is a language for object-oriented programming.   
  
Classes and objects, together with their characteristics and methods, are linked to everything in C++. For instance, an automobile is an object in the actual world. The car contains features like color and weight, as well as functions like brake and drive.  
  
In essence, variables and functions that are part of the class are what attributes and methods are. The term "class members" is frequently used to describe this.   
  
In our application, a class is a user-defined data type that serves as an object constructor, or "blueprint" for object creation.

**Que 2. What is class in C++ and how does it differ from an object?**

**Ans: Class in C++:** A class in C++ is a blueprint or template for creating objects. It defines a datatype by bundling data and methods that operate on the data into a single unit. Here’s a simple example:

|  |
| --- |
| class Car {  public:  // Attributes  string brand;  string model;  int year;  // Methods  void displayInfo() {  cout << "Brand: " << brand << ", Model: " << model << ", Year: " << year << endl;  }  }; |

In this example, Car is a class with attributes brand, model, and year, and a method displayInfo().

**Object in C++**

An **object** is an instance of a class. When you create an object, you allocate memory for it and can use the class's methods and attributes. Here’s how you create an object from the Car class:

|  |
| --- |
| int main() {  // Creating an object of Car class  Car myCar;  // Setting attributes  myCar.brand = "Toyota";  myCar.model = "Corolla";  myCar.year = 2020;  // Using method  myCar.displayInfo();  return 0;  } |

In this example, myCar is an object of the Car class. It has its own set of attributes and can use the displayInfo() method defined in the class.

Key Differences

1. Definition vs. Instance:

* Class: Defines the structure and behavior (attributes and methods).
* Object: An instance of the class with actual values assigned to the attributes.

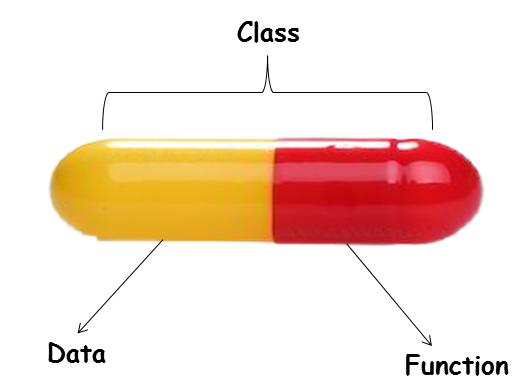
1. Memory Allocation:

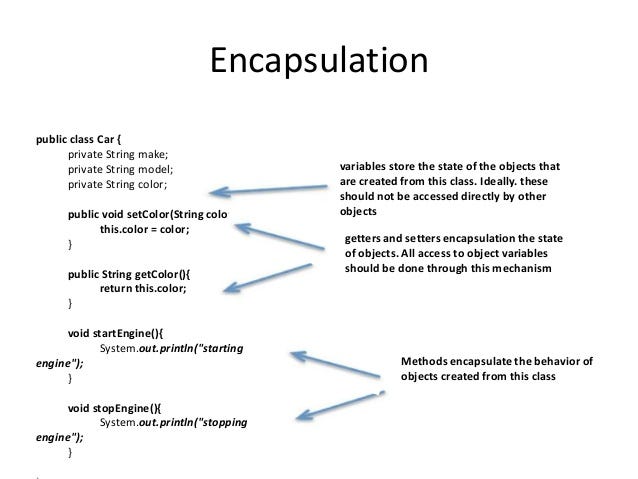
* Class: No memory is allocated when a class is defined.
* Object: Memory is allocated when an object is created.

1. Usage:

* Class: Used to define the blueprint.
* Object: Used to interact with the blueprint, performing operations and storing data.

Que 3. Explain the concept of encapsulation with an example.

Ans: Encapsulation is a process of combining member functions and data members in a single unit called a class. The purpose is to prevent access to the data directly. Access to them is provided through the functions of the class. It is one of the popular features of Object-Oriented Programming(OOPs), which helps in **data hiding**.



**Que 4: How do you define a class in C++?**

**Ans:** **A** class is a user-defined data type that encapsulates data and functions. It is defined using the class keyword followed by the class name and a block containing its members.

|  |
| --- |
| class ClassName {  access\_specifier:  // Data members  // Member functions  } object\_names; |

**Key Points:**

* Members can be **private**, **protected**, or **public**.
* By default, class members are **private**.
* Objects access public members using the dot (.) operator.

**Que 5. Describe the syntax for creating an object of a class.**  
**Ans:** To create an object of a class, use the class name followed by the object name.

|  |
| --- |
| class MyClass {  // class members  };  MyClass obj; // object creation |

Here, obj is an object of type MyClass.

**Que 6. What are private members in a class and how are they accessed?**

**Ans:** Private members in a class are members that are not accessible from outside the class. They can only be accessed within the class itself.

e.g., In C++, we can declare private members using the private access specifier.

syntax:

class ClassName {

    private:

    // Private members of the class

};

int main() {

    ClassName obj;

    obj.member1 = 10; // This will not compile as member1 is a private member

    return 0;

}

**Que 7. What are public members in a class and how are they accessed?**

**Ans:** Public members of a class are accessible from outside the class.

They can be accessed using the object name followed by the member name.

e.g.,

|  |
| --- |
| class Person {  // Class declaration  public:     // Public access specifier      string name;      int age;      void setPersonDetails(string n, int a) {   // Method to set name and age          name = n;          age = a;      }      void showPersonDetails() {   //     Method to display name and age          cout << "Name: " << name << endl;          cout << "Age: " << age << endl;      }  }  int main() {  // Main function      Person p;       // Creating object of Person class      p.setPersonDetails("Sanskrati Shukla", 20);     // Setting name and age      p.showPersonDetails();          // Displaying name and age      return 0;  } |
| **// output:**  **// Name: Sanskrati Shukla**  **// Age: 20** |

**Que 8. Explain the significance of access specifiers in a class.**

**Ans:** Access specifiers in a class determine the accessibility of class members (variables, methods, etc.) to other classes. There are three access specifiers in C++: public, private, and protected.

1. **Public access specifier:**

The public access specifier allows access to class members from any other class.

1. **Private access specifier:**

The private access specifier restricts access to class members from outside the class.

1. **Protected access specifier:**

The protected access specifier allows access to class members from derived classes.

**Syntax:**

class ClassName {

    public:

        // Public members

    private:

        // Private members

    protected:

        // Protected members

};

int main() {

    // Creating an object of MyClass

    MyClass obj;

    // Accessing public member

    obj.publicMember = 10;

    // Attempting to access private member

    // obj.privateMember = 20; // Error: Private member is not accessible

    // Attempting to access protected member

    // obj.protectedMember = 30; // Error: Protected member is not accessible

    return 0;

}

// Output:

// 10

e.g., In the above example, we have a class MyClass with public, private, and protected members.

The public member is accessible from any other class, while the private and protected members are not accessible.

**Que 9. Provide an example of a class with both private and public members.**

**Ans:**

class MyClass:      // MyClass is a class

    def \_\_init\_\_(self):     // \_\_init\_\_ is a constructor method

        self.\_private\_member = 10       // Private member of the class

        self.public\_member = 20         // Public member of the class

    def access\_private\_member(self):    // Method to access private member

        return self.\_private\_member     // Returns the private member

    def access\_public\_member(self):     // Method to access public member

        return self.public\_member       // Returns the public member

my\_object = MyClass()                   // Creating an object of MyClass

print(my\_object.access\_private\_member())  # Output: 10

print(my\_object.access\_public\_member())   # Output: 20

**Que 10. How does data hiding work in C++?**

**Ans:** Data hiding in C++ is a mechanism that prevents direct access to internal data members of a class.

It is achieved by declaring the data members as private and providing access to them through public member functions.

In C++, data hiding is achieved by declaring data members as private and providing access to them through public member functions.

This prevents direct access to the data members and ensures that the class is used correctly.

**Syntax:**

// Class declaration

class ClassName {

private:

    // Private data members

public:

    // Public member functions

};

int main() {

    ClassName object;  // Creating an object of the ClassName class

    object.publicMemberFunction(); // Accessing public member function

    return 0;  // Returning 0 to indicate successful execution

}

**Que 11. What is a static data member in C++?**

**Ans:** A static data member is a member of a class that is declared as static. It is a class variable that is shared by all objects of the class.

The static data member is initialized once, when the class is loaded into memory, and is not destroyed when the objects of the class are destroyed.

Static data members are useful when you want to maintain a single copy of a variable that is shared by all objects of a class. e.g.,

#include <iostream>

class MyClass {

public:

    static int count;

};

int MyClass::count = 0;

int main() {

    MyClass obj1;

    // Accessing static data member

    std::cout << MyClass::count << std::endl;

    return 0;

}

**Que 12. How do you declare and initialize a static data member?**

**Ans:** The declaration and initialization of static data members take place outside of the class specification.   
Global variables that are shared by all instances of a class are usually represented using static data members.

static int count = 0;

**Que 13. What is a static function member in C++?**

**Ans:** A static function member is a function that is associated with a class rather than an object.

    It is declared using the static keyword and belongs to the class rather than the object.

**Syntax:**

|  |
| --- |
| class ClassName {          public:              static void functionName() {                  // Function body              }      };      Usage:      ClassName::functionName();      Example:      class MyClass {          public:              static void printMessage() {                  cout << "Hello, world!" << endl;              }      };      int main() {          MyClass::printMessage();          return 0;      } |

**Que** **14. How do static function members differ from regular function members?**  
**Ans:** Static functions belong to the class, not objects.  
They arew:

* Can be called without creating an object
* Only access **static** members
* Don’t have access to this pointer

  ClassName::staticFunction(); // called directly

**Que 15. Provide an example of a class with static data and function members.  
Ans:**

|  |
| --- |
| class Counter {  public:  static int count;  static void showCount() {  cout << "Count: " << count << endl;  }  };  int Counter::count = 0; |

**Que** **16. What is a constructor in C++ and why is it important?**  
**Ans:** A constructor is a special function that is **automatically called** when an object is created.  
It initializes the object's data members.

**Key points:**

* Same name as the class
* No return type (not even void)
* Helps in setting up an object with initial values

|  |
| --- |
| class Demo {  public:  Demo() {  cout << "Constructor called!" << endl;  }  }; |

**Que 17. Explain the different types of constructors in C++.  
Ans:** C++ supports several constructor types:

1. **Default Constructor –** Takes no arguments.
2. **Parameterized Constructor –** Takes parameters to initialize data.
3. **Copy Constructor –** Initializes an object using another object of the same class.
4. **Constructor with Default Arguments –** Parameters have default values.
5. **Delegating Constructor (C++11) –** One constructor calls another in the same class.

Each type helps with different object initialization needs.

**Que 18. What is a default constructor and when is it used?  
Ans:** A default constructor is a constructor that takes no parameters.  
It's automatically called when an object is created without any arguments.

**Use case:**To initialize objects with default values.

|  |
| --- |
| **class MyClass {**  **public:**  **int x, y;**  **MyClass() { // Default constructor**  **x = 0; y = 0;**  **}**  **};**  **int main() {**  **MyClass obj; // Default constructor called**  **}** |

**Que 19. How do parameterized constructors work?**

**Ans:** A parameterized constructor in C++ is a type of constructor that takes arguments (parameters). Giving arguments allows you to initialize an object with specific values when you create it instead of always providing the object's default values.

**Usage**:

* Allows passing values to the constructor at object creation.
* More flexible than default constructors.

**Syntax**:

ClassName(parameter1, parameter2) {

// Initialize members with parameters

}

|  |
| --- |
| class MyClass {  public:  int x, y;  // Parameterized constructor  MyClass(int a, int b) {  x = a;  y = b;  }  };  int main() {  MyClass obj(10, 20); // Calls parameterized constructor  cout << "x = " << obj.x << ", y = " << obj.y << endl;  } |

**Que 20. What is a copy constructor and what is its purpose?**

**Ans:** A **copy constructor** in C++ is a special type of constructor used to create a new object as a copy of an existing object of the same class. It initializes an object using another object of the same class, performing a member-by-member copy.

**Syntax:**

ClassName(const ClassName& other);

|  |
| --- |
| class MyClass {  public:  int x;  MyClass(int val) { x = val; }  // Copy constructor  MyClass(const MyClass& obj) {  x = obj.x;  }  };  int main() {  MyClass a(5);  MyClass b = a; // Copy constructor called  cout << b.x; // Output: 5  } |

**Que 21. Explain the concept of constructor overloading ?**

**Ans:** Constructor overloading in C++ allows a class to have multiple constructors with the same name but different parameter lists. This feature enhances the flexibility and usability of a class by enabling the creation of objects with varied initialization options.

**Key Principles**

In C++, constructors can be overloaded in a similar way to function overloading. Overloaded constructors have the same name as the class but differ in the number and type of arguments. The appropriate constructor is called based on the arguments passed during object creation.

**Code Example**

|  |
| --- |
| class Box {  public:  int length;  Box() { length = 0; } // Default constructor  Box(int l) { length = l; } // Parameterized constructor  };  int main() {  Box a; // Calls default constructor  Box b(10); // Calls parameterized constructor  cout << b.length; // Output: 10  } |

**Que 22. How does a constructor initializer list work?**

**Ans:** **A** constructor initializer list is a feature that allows you to initialize member variables of a class before the constructor's body executes. This is particularly useful for initializing constant members, reference members, and base class members.

**Syntax and Usage**

The initializer list is defined after the constructor's parameter list and begins with a colon (:), followed by a comma-separated list of initializations. Each initialization consists of a member name followed by its initialization value enclosed in braces {} or parentheses ().

ClassName(type val) : member(val) { }

**Use Cases:**

* For initializing **const**, **reference** members, or base class constructors.
* More **efficient** than assigning in the body.

E.g.,

class Point {

int x, y;

public:

Point(int a, int b) : x(a), y(b) { }

};

**Que 23. What is a destructor in C++ and what is its purpose?**

**Ans:** A **destructor** is a special member function in C++ that is automatically invoked when an object goes out of scope or is explicitly deleted. Its primary purpose is to free resources that the object may have acquired during its lifetime, such as memory, file handles, or network connections.

**Syntax and Characteristics**

The syntax for defining a destructor is straightforward. It has the same name as the class, preceded by a tilde (~) symbol, and it does not take any arguments or return any value:

class MyClass {

public:

~MyClass() {

// Destructor code

}

};

Destructors have several key characteristics:

* **Unique**: A class can have only one destructor.
* **Non-overloadable**: Destructors cannot be overloaded.
* **Automatic Invocation**: They are called automatically when an object goes out of scope or is explicitly deleted.

**Que 24. How is a destructor declared and defined?**

**Ans:** A **destructor** is declared with a **tilde (~)** followed by the class name. It takes **no parameters** and has **no return type**.

**Syntax:**

|  |
| --- |
| class ClassName {  public:  ~ClassName(); // Declaration  };  ClassName::~ClassName() { // Definition  // Cleanup code  } |

**Que 25. What happens if a destructor is not explicitly defined in a class?**

**Ans:** If a destructor is **not explicitly defined**, C++ automatically provides a **default destructor** that performs **shallow cleanup**, such as destroying built-in types.

But if your class allocates **dynamic memory**, you should define a destructor to prevent **memory leaks**.

**Que 26. Explain the concept of automatic and dynamic storage duration in relation to destructors.**

**Ans:** 1. Automatic storage duration: Blocks or functions are used to generate objects. The moment they leave their scope, their destructor is instantly triggered.

2. Duration of dynamic storage: Items made with new must be manually removed with delete. When you destroy an object explicitly, the destructor is called.

Memory leaks occur when deletion is forgotten.

**Que 27. How do destructors differ from constructors?**

**Ans:** **Constructor**:

* Called when an object is **created**.
* Initializes the object's **data members**.
* Can have **parameters** and be overloaded.

**Destructor**:

* Called when an object is **destroyed**.
* Cleans up resources and performs **memory deallocation**.
* **No parameters** and cannot be overloaded.

**Que 28. What is operator overloading in C++ and why is it useful?**

**Ans:** You can change how operators (such as +, -, etc.) behave for user-defined types (classes) by using operator overloading. Because operators may be applied to custom objects in the same manner as they are to built-in types, this improves the readability and intuitiveness of the code.

|  |
| --- |
| class Complex {  public:  int real, imag;  Complex operator + (const Complex& obj) {  Complex temp;  temp.real = real + obj.real;  temp.imag = imag + obj.imag;  return temp;  }  }; |

**Usefulness:**

* **Custom behavior** for common operations.
* Enhances code **clarity** and **maintainability**.

**Que 29. Describe the syntax for overloading an operator.**

**Ans:** In C++, overloading an operator involves defining a function that employs the operator keyword, operator symbol, and relevant parameters.

|  |
| --- |
| class Complex {  public:  int real, imag;    Complex operator + (const Complex& obj) {  Complex temp;  temp.real = real + obj.real;  temp.imag = imag + obj.imag;  return temp;  }  }; |

**Que 30. Which operators can and cannot be overloaded in C++?**

**Ans:** In C++, most operators can be overloaded, but there are some exceptions.

Operators that can be overloaded:

* Arithmetic operators (+, -, \*, /, etc.)
* Comparison operators (==, !=, <, >, etc.)
* Assignment operators (=, +=, -=, etc.)
* Increment and Decrement operators (++, --)
* Stream insertion and extraction (<<, >>)

Operators that cannot be overloaded:

* :: (Scope resolution operator)
* . (Member access operator)
* .\* (Pointer-to-member operator)
* ?: (Ternary conditional operator)
* sizeof (Sizeof operator)

**Que 31. Provide an example of overloading the "+" operator for a custom class.**  
**Ans:** Here's an example of overloading the + operator for a Complex class:

|  |
| --- |
| class Complex {  public:  int real, imag;  Complex operator + (const Complex& obj) {  Complex temp;  temp.real = real + obj.real;  temp.imag = imag + obj.imag;  return temp;  }  };  // Output:  //Complex c1, c2, result;  //c1.real = 1; c1.imag = 2;  //c2.real = 3; c2.imag = 4;  //result = c1 + c2; // Uses overloaded //"+" operator |

**Que 32. Explain the concept of friend functions in the context of operator overloading.  
Ans:** A class's private and protected members can be accessed by the friend function, a non-member function. When an operator needs access to the private information of two objects from distinct classes but is not a member function of either class, it can overload operators by using a friend function.

**Que 33. What is a friend function in C++ and how is it declared?**  
**Ans:**  
A **friend function** is a function that is allowed to access the private and protected members of a class. It is declared inside the class using the friend keyword.

**Syntax:**

class ClassName {

friend return\_type function\_name();

};

**Que 34. How do friend functions differ from member functions?**  
**Ans:**

* **Member functions** are defined within the class and can directly access all members (public, private, and protected) of the class.
* **Friend functions** are not members of the class but can still access the private and protected members of the class due to the friend keyword. They can be regular functions or even other classes’ functions.

**Que 35. Explain the benefits and potential drawbacks of using friend functions.**  
**Ans:**  
**Benefits:**

* Friend functions can access private and protected data, making them useful for operations that require access to these members.
* They allow operator overloading and other operations that may need to access private members without making them public.

**Drawbacks:**

* Overusing friend functions can break encapsulation and lead to maintenance issues.
* Since they are not part of the class, they don't enjoy the benefits of member functions (e.g., inheritance).

**Que 36. What is inheritance in C++ and why is it important?**  
**Ans: A** key component of OOP is inheritance, which enables a class (derived class) to inherit attributes and methods from another class (base class).   
It helps the development of specialized classes, permits hierarchical class hierarchies, and encourages code reuse.

|  |
| --- |
| class Animal {  public:  void eat() { cout << "Eating"; }  };  class Dog : public Animal { // Dog inherits from Animal  public:  void bark() { cout << "Barking"; }  }; |

**Que 37. Explain the different types of inheritance in C++.**  
**Ans:** C++ supports the following types of inheritance:

* **Single inheritance**: A derived class inherits from one base class.
* **Multiple inheritance**: A derived class inherits from two or more base classes.
* **Multilevel inheritance**: A derived class inherits from another derived class.
* **Hierarchical inheritance**: Multiple classes inherit from a single base class.
* **Hybrid inheritance**: A mix of two or more types of inheritance.

**Que 38. How do you implement single inheritance in C++?**  
**Ans:**  
In **single inheritance**, a derived class inherits from a single base class.

**Example:**

|  |
| --- |
| class Animal {  public:  void eat() { cout << "Eating"; }  };  class Dog : public Animal { // Dog inherits from Animal  public:  void bark() { cout << "Barking"; }  }; |

**Que 39. What is multiple inheritance and how does it differ from single inheritance?**  
**Ans:**  
**Multiple inheritance** allows a derived class to inherit from more than one base class. It differs from single inheritance, where a class can only inherit from one base class.

**Example of Multiple Inheritance:**

|  |
| --- |
| class Animal {  public:  void eat() { cout << "Eating"; }  };  class Pet {  public:  void play() { cout << "Playing"; }  };  class Dog : public Animal, public Pet { // Dog inherits from both Animal and Pet  public:  void bark() { cout << "Barking"; }  }; |

**Que 40. Describe hierarchical inheritance with an example.**  
**Ans:**  
**Hierarchical inheritance** occurs when multiple derived classes inherit from a single base class.

**Example:**

|  |
| --- |
| class Animal {  public:  void eat() { cout << "Eating"; }  };  class Dog : public Animal {  public:  void bark() { cout << "Barking"; }  };  class Cat : public Animal {  public:  void meow() { cout << "Meowing"; }  }; |

**Que 41. What is multilevel inheritance and how is it implemented in C++?**  
**Ans:**  
In **multilevel inheritance**, a derived class inherits from another derived class.

**Example:**

|  |
| --- |
| class Animal {  public:  void eat() { cout << "Eating"; }  };  class Dog : public Animal {  public:  void bark() { cout << "Barking"; }  };  class Puppy : public Dog { // Puppy inherits from Dog, which inherits from Animal  public:  void play() { cout << "Playing"; }  }; |

**Que 42. Explain the concept of hybrid inheritance.**  
**Ans:**  
**Hybrid inheritance** is a combination of two or more types of inheritance, such as single, multiple, or multilevel inheritance.

**Example:**

|  |
| --- |
| class Animal {  public:  void eat() { cout << "Eating"; }  };  class Pet {  public:  void play() { cout << "Playing"; }  };  class Dog : public Animal, public Pet { // Hybrid inheritance  public:  void bark() { cout << "Barking"; }  }; |

**Que 43. What are access modifiers in C++ and what are the different types?**  
**Ans:**  
**Access modifiers** control the visibility and accessibility of class members.  
There are three main types:

* **public**: Members are accessible from outside the class.
* **private**: Members are accessible only within the class.
* **protected**: Members are accessible within the class and derived classes.

**Que 44. How do public, private, and protected access modifiers affect inheritance?**  
**Ans:**

* **Public** members of a base class are inherited as public members in the derived class.
* **Private** members of a base class are not accessible directly in the derived class.
* **Protected** members of a base class are inherited as protected members in the derived class.

**Que 45. Explain how access modifiers control member accessibility in derived classes.**  
**Ans:**  
In derived classes:

* **Public members** remain **public**.
* **Private members** cannot be accessed directly by derived classes.
* **Protected members** remain **protected** and can be accessed by derived classes.

**Que 46. What is function overriding in the context of inheritance?**  
**Ans:** When a derived class offers a particular implementation for a function that is already specified in its base class, this is known as function overriding. The derived class function must have the same signature as the base class function, which must be designated as virtual.

**Que 47. How do you override a base class function in a derived class?**  
**Ans:** To override a function, use the virtual keyword in the base class and define a new version of the function with the same signature in the derived class.

**Example:**

|  |
| --- |
| class Animal {  public:  virtual void sound() { cout << "Animal sound"; }  };  class Dog : public Animal {  public:  void sound() override { cout << "Barking"; } // Overridden function  }; |

**Que 48. Explain the use of the "virtual" keyword in function overriding.**  
**Ans:** The base class function indicates that it can be overridden in derived classes by using the virtual keyword. It permits runtime polymorphism by guaranteeing that the appropriate function (base or derived) is invoked at runtime.

**Que 49. What is the significance of the "override" specifier in C++11 and later?**  
**Ans:** A function's actual overriding of a base class function is guaranteed by the override specifier. To help prevent bugs, the compiler will produce an error if the base class contains no matching function.

**Que 50. What is a virtual base class in C++ and why is it used?**  
**Ans:** In multiple inheritance, a virtual base class is used to guarantee that, even in cases when a base class is inherited more than once, the derived class only inherits one copy of the base class.

**Que 51. How do you declare and implement a virtual base class?**  
**Ans:** When defining the base class in the derived class, use the virtual keyword to establish a virtual base class**.**

**Example:**

|  |
| --- |
| class A {  public:  void display() { cout << "Class A"; }  };  class B : virtual public A {}; // Virtual inheritance from A  class C : virtual public A {}; // Virtual inheritance from A  class D : public B, public C {}; // D inherits A only once |

**Que 52. Explain the role of virtual base classes in resolving ambiguity in multiple inheritance.**  
**Ans:** There may be uncertainty when attempting to access members of the base class when two classes share a common base class through multiple inheritance. By ensuring that only one instance of the base class is inherited, a virtual base class helps to eliminate ambiguity.

**Que 53. Provide an example of using a virtual base class to avoid the diamond problem in inheritance.**  
**Ans:** When two classes inherit from the same base class and a third class inherits from both, ambiguity results, which is known as the "**diamond problem**." This is fixed by using virtual inheritance.

|  |
| --- |
| class A {  public:  void display() { cout << "Class A"; }  };  class B : virtual public A {}; // Virtual inheritance  class C : virtual public A {}; // Virtual inheritance  class D : public B, public C {}; // Only one instance of A is inherited  int main() {  D d;  d.display(); // Calls display() from A  } |