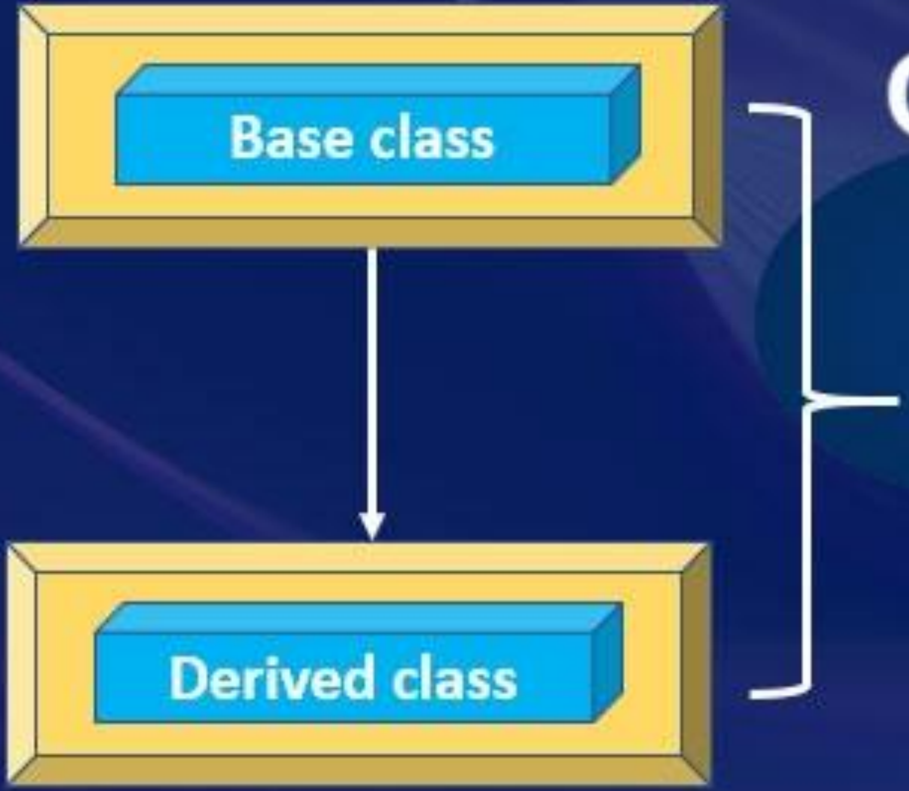
Concept of Reusability



C++

By – Mohammad Imran

Concept of Reusability

Introduction

C++ strongly supports the concept of reusability. The C++ classes can be reused in several ways. Once a class has been written and tested, it can be adapted by another programmer to suit their requirements. This is basically done by creating new classes, reusing the properties of the existing ones. The mechanism of deriving a new class from an old one is called inheritance. The old class is referred to as the base class and the new one is called the derived class or subclass. A derived class includes all features of the generic base class and then adds qualities specific to the derived class.

Concept of Reusability

Introduction

Code reuse is a fundamental principle in OOP that aims to reduce redundancy and improve maintainability by using existing code in new contexts. The idea is to write code once and use it multiple times across different parts of a program or in different programs.

C++ features such as classes, virtual function, and templates allow designs to be expressed so that re-use is made easier there are many advantage of reusability. They can be applied to reduce cost, effort and time of software development. It also increases the productivity, portability and reliability of the software product.

By – Mohammad Imran

Defining Derived Class

In OOP, a **derived class** (or subclass) is a class that is based on another class, known as the **base class** (or superclass). The derived class inherits attributes and methods from the base class, allowing it to reuse and extend the functionality of the base class.

Defining Derived Class

Syntax

In most object-oriented programming languages, you define a derived class using a syntax that specifies the base class from which it inherits.

```
class class_Name1
{
   //Data Member
   //Member Function
}
class class_Name2 : Access_Specifier class_Name1
{
   //Data Member
   //Member Function
}
```

By – Mohammad Imran

Defining Derived Class

Example - 1

```
#include <iostream>
using namespace std;
class Base
{
  protected:
   int value;
  public:
   Base(int value) : value(value)
   {}
   void show()
   {
     cout << "Base Value: " << value << endl;
   }
};
```

By – Mohammad Imran

Defining Derived Class

Example - 1

```
class Derived : public Base
{
  public:
   Derived(int value) : Base(value)
   {}
   void display()
   {
     cout << "Derived class value: " << value << endl;
   }
};
```

OUTPUT

```
100
100
```

```
int main()
{
   Derived d(100);
   d.show();
   d.display();
}
```

By – Mohammad Imran

Inheritance

Introduction

In C++, inheritance enables an object to automatically acquire all properties and behaviors from its parent object, allowing for the reuse, extension, or modification of attributes defined in another class. The class inheriting these members is the derived class, while the class providing the inherited members is the base class. This capability, a key aspect of Object-Oriented Programming, involves creating new classes ("derived" or "child" classes) from existing classes ("base" or "parent" classes), fostering code reusability and flexibility.

Inheritance

Why and When to Use

We use inheritance in C++ for the reusability of code from the existing class. C++ strongly supports the concept of reusability. Reusability is yet another essential feature of OOP(Object Oriented Programming).

It is always good to reuse something that already exists rather than trying to create the one that is already present, as it saves time and increases reliability.

We use inheritance in C++ when both the classes in the program have the same logical domain and when we want the class to use the properties of its superclass along with its properties.

Inheritance Mode

There are three modes of inheritance:

- ✓ Public mode
- ✓ Protected mode
- ✓ Private mode

Inheritance

Public Mode

In the public mode of inheritance, when a child class is derived from the base or parent class, then the public member of the base class or parent class will become public in the child class also, in the same way, the protected member of the base class becomes protected in the child class, and private members of the base class are not accessible in the derived class.

Inheritance

Protected Mode

In protected mode, when a child class is derived from a base class or parent class, then both public and protected members of the base class will become protected in the derived class, and private members of the base class are again not accessible in the derived class. In contrast, protected members can be easily accessed in the derived class.

By – Mohammad Imran

Inheritance

Private Mode

In private mode, when a child class is derived from a base class, then both public and protected members of the base class will become private in the derived class, and private members of the base class are again not accessible in the derived class.

Inheritance Access Table

Base Class Access Specifier	Type of Inheritance		
	Public	**Protected**	**Private**
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not Accessible	Not Accessible	Not Accessible

By – Mohammad Imran

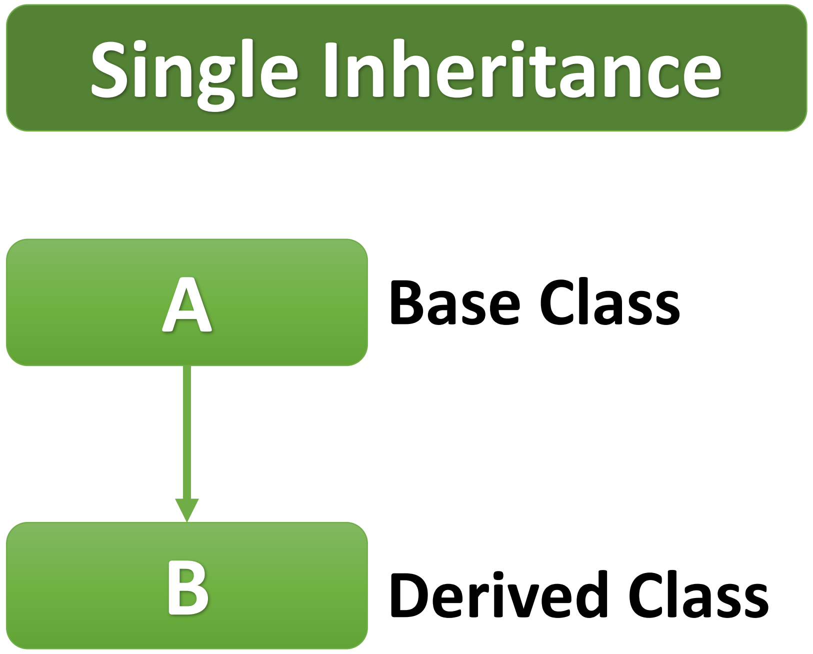
Inheritance Types

There are five types of inheritance in C++:

- ✓ Single Inheritance
- ✓ Multiple Inheritance
- ✓ Multilevel Inheritance
- ✓ Hybrid Inheritance
- ✓ Hierarchical Inheritance

Single Inheritance

When the derived class inherits only one base class, it is known as Single Inheritance.



In the above diagram, A is a Base class, and B is a derived class. Here the child class inherits only one parent class.

By – Mohammad Imran

Single Inheritance

Example - 2

```
#include<iostream>
using namespace std;
class Base
{
   public:
    float salary;
    void basic()
    {
        salary = 900;
    }
};
```

```
class Derived : public Base
{
   public:
    float bonus;
    void gross()
    {
        bonus = 100;
    }
    void sum()
    {
        cout << "Total Salary = ";
        cout << (salary + bonus) << endl;
    }
};
```

By – Mohammad Imran

Single Inheritance

Example - 2

```
int main()
{
   Derived d;
   d.basic();
   d.gross();
   cout << "Salary = " << d.salary << endl;
   cout << "Bonus = " << d.bonus << endl;
   d.sum();
   return 0;
}
```

OUTPUT

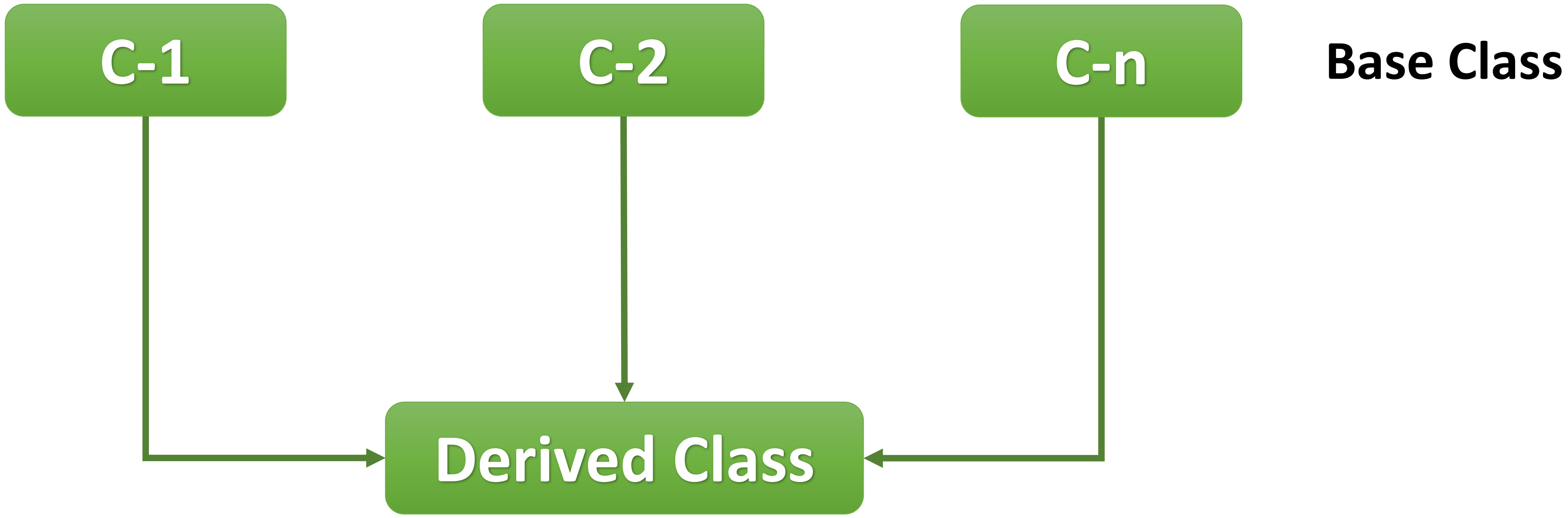
```
Salary = 10000
Bonus  = 2000
Total Salary = 12000
```

Multiple Inheritance

Multiple inheritance is a concept used in object-oriented programming where we can inherit attributes and behaviors from more than one parent class to a given child class. In languages that support multiple inheritances, like C++ programming, a derived class can have multiple base classes, and it inherits data members and member functions from all of them..

By – Mohammad Imran

Multiple Inheritance



In the above diagram, C-1, C-2 and C-n is a Base class, and Derived is a derived class. Here the child class inherits more than one parent class.

By – Mohammad Imran

Multiple Inheritance

Example - 3

Let's consider a vehicle with a navigation system.

Imagine creating a class for a vehicle that can inherit features from both a Car class and a GPS class. The Car class might provide information about speed and fuel consumption, while the GPS class offers location and navigation capabilities. By using multiple inheritance, we can create a SmartCar class that combines the functionalities of a car and a GPS system.

Multiple Inheritance

Example - 3

```
#include <bits/stdc++.h>
using namespace std;
class Car //Base Class
{
  public:
   void drive()
   {
      cout << "Car is driving." << endl;
   }
};
class GPS //Base Class
{
  public:
   void navigate()
   {
      cout << "GPS is navigating." << endl;
   }
};
```

By – Mohammad Imran

Multiple Inheritance

Example - 3

```
class SmartCar : public Car, public GPS
{
  public:
   void autoDrive()
   {
      cout << "Auto-Driving SmartCar." << endl;
   }
};
int main()
{
   SmartCar myCar;
   myCar.drive();
   myCar.navigate();
   myCar.autoDrive();
   return 0;
}
```

By – Mohammad Imran

Multiple Inheritance | Example - 3 | Explanation

We begin the C++ program by including the bits/stdc++.h file to access all standard header files. Then-

1. We create a ***base class*** called ***Car***, which contains a ***drive() member function***. This function uses std::cout to print a phrase when called.
2. Then, we define a ***second base class*** called ***GPS***, which also contains a ***member function navigate()*** to print a phrase when called.
3. Next, we define a ***derived class*** named ***SmartCar*** using multiple inheritance. It inherits from both the Car and GPS classes.

Multiple Inheritance | Example - 3 | Explanation

4. This means that SmartCar inherits the member functions and attributes of both base classes. It also has a member function, **autoDrive()**, which prints a phrase using std::cout.
5. Inside the **main() function**, we-
   - Create an instance of the SmartCar class called **myCar**.
   - We use this object and the **dot operator** to **call the drive() function** inherited from the Car class and **call the navigate() function** inherited from the GPS class.

Multiple Inheritance | Example - 3 | Explanation

- These print the phrases, 'Car is driving', and 'GPS is navigating', respectively.
- Lastly, we call the **autoDrive()** unique member function of the **SmartCar** class using the dot operator. This prints the phrase 'SmartCar is auto-driving' to the console.

6. Finally, the program completes execution with a return 0 statement.

Multiple Inheritance | Arithmetic Operation | Example - 4

```
#include <bits/stdc++.h>
using namespace std;
class Addition
{
  public:
   int add(int a, int b)
   {
      return a + b;
   }
};
class Subtraction
{
  public:
   int subtract(int a, int b)
   {
      return a - b;
   }
};
```

By – Mohammad Imran

Multiple Inheritance

Arithmetic Operation **Example - 4**

```
class Calculator : public Addition, public Subtraction
{
   public:
    Calculator()
    {
       cout << "Calculator is ready." << endl;
    }
    void performOperations(int num1, int num2)
    {
       int sum = add(num1, num2);
       int sub = subtract(num1, num2);
       cout << "Addition result: " << sum << endl;
       cout << "Subtraction result: " << sub << endl;
    }
};
```

By – Mohammad Imran

Multiple Inheritance | Arithmetic Operation | Example - 4

```
int main()
{
   Calculator myCalculator;
   int num1 = 10, num2 = 5;
   myCalculator.performOperations(num1, num2);
   return 0;
}
```

By – Mohammad Imran

Multiple Inheritance | Example - 4 | Explanation

✓ We create a **base class** called **Addition**, which contains a public member function called **add()**. This function tables two integers, a and b, and returns their sum using the **addition arithmetic operator**.

✓ Then, we define a **second base class** called **Subtraction** with public member function **subtract()** that takes two integers and returns their difference.

Multiple Inheritance | Example - 4 | Explanation

✓ Next, we create a **derived class** called **Calculator**, which publically inherits from both the Addition and Subtraction classes. Inside this class-

- There is a constructor called **Calculator()**, which prints the phrase 'Calculator is ready' when a class object is created.
- We define a function, **performOperations()**, which performs both addition and subtraction using the inherited add and subtract methods.
- It takes two integer numbers as input and calculates their addition (stored in **resultAddition**) and subtraction (stored in **resultSubtraction**). The result is printed using cout and endl, which moves the cursor to the next line.

By – Mohammad Imran

Multiple Inheritance | Example - 4 | Explanation

✓ In the **main() function**, we-

- Create an instance of the Calculator class named **myCalculator**.
- Declare and initialize two integer type variables, **num1** and **num2**, with values **10** and **5** respectively.
- Nest, we **call the performOperations()** function on myCalculator object and pass num1 and num2 as parameters.
- The function uses the methods from derived classes to perform the operations and displays the results of addition and subtraction to the console.

✓ The program terminates with a return 0.

Getting Average Marks Of 2 Subjects Using Multiple Inheritance In C++

In this example, we'll create a program with two base classes, i.e., Maths and Science, representing marks obtained in math and science subjects, respectively. Then, we'll create a derived class Student that inherits from both Math and Science to calculate and display the average marks.

NOTE: This program also demonstrate the Base Class Initialization. Means we can initialize member variable of all base class.

Multiple Inheritance

Example - 5

```
#include <bits/stdc++.h>
using namespace std;
// Base class for math subject
class Maths
{
  public:
   double mathMarks;
   Maths(double math)  : mathMarks(math)
   {}
   void displayMathMarks()
   {
      cout << "Math Marks: " << mathMarks << endl;
   }
};
```

```
Maths(double math)
{
   mathMarks(math)
}
```

By – Mohammad Imran

Multiple Inheritance

Example - 5

```
// Base class for science subject
class Science
{
  public:
   double scienceMarks;
   Science(double science) : scienceMarks(science)
   {}
   void displayScienceMarks()
   {
      cout << "Science Marks: " << scienceMarks << endl;
   }
};
```

Multiple Inheritance

Example - 5

```
// Derived class Student inherits from both Maths and Science

class Student : public Maths, public Science
{
  public:
   Student(double math, double science) : Maths(math),
   Science(science)
   {}
   void calculateAverage()
   {
      double average = (mathMarks + scienceMarks) / 2.0;
      cout << "Average Marks: " << average << endl;
   }
};
```

By – Mohammad Imran

Multiple Inheritance

Example - 5

```
int main()
{
   double mathMarks = 90.5;
   double scienceMarks = 88.0;
   Student student(mathMarks, scienceMarks);
   student.displayMathMarks();
   student.displayScienceMarks();
   student.calculateAverage();
   return 0;
}
```

By – Mohammad Imran

Multilevel Inheritance

Multilevel inheritance is a type of inheritance in C++ where a derived class inherits from a base class, and the base class itself inherits from another base class. This forms a hierarchical relationship between classes, where a class can inherit properties and behavior from multiple levels of base classes.

In multilevel inheritance, a derived class inherits from a base class, which is itself a derived class from another base class. This creates a chain of inheritance, where the derived class inherits properties and behavior from multiple levels of base classes.

By – Mohammad Imran

Multilevel Inheritance

Syntax

```
class GrandParent
{
   // members of GrandParent
};
class Parent : public GrandParent
{
   // members of Parent
};
class Child : public Parent
{
   // members of Child
};
```

By – Mohammad Imran

Multilevel Inheritance

Example - 6

```
#include <iostream>
#include <string>
using namespace std;
class University
{
  public:
   string name;
   University(string n) : name(n)
   {}
   void displayUniversity()
   {
      cout << "University: " << name << endl;
   }
};
```

By – Mohammad Imran

Multilevel Inheritance

Example - 6

```
class Department : public University
{
  public:
   string departmentName;
   Department(string n, string dn) : University(n),
   departmentName(dn)
   {}
   void displayDepartment()
   {
      cout << "Department: " << departmentName << endl;
   }
};
```

By – Mohammad Imran

Multilevel Inheritance

Example - 6

```
class Course : public Department
{
  public:
   string courseName;
   Course(string n, string dn, string cn) : Department(n, dn),
   courseName(cn)
   {}
   void displayCourse()
   {
      cout << "Course: " << courseName << endl;
   }
};
```

By – Mohammad Imran

Multilevel Inheritance

Example - 6

```
int main()

{

   Course myCourse("Harvard", "Computer Science", “C++");

   myCourse.displayUniversity();

   myCourse.displayDepartment();

   myCourse.displayCourse();

   return 0;

}
```

OUTPUT

```
Harvard
Computer Science
C++
```

By – Mohammad Imran

Ambiguity in Inheritance

Ambiguity in inheritance occurs when a derived class inherits conflicting attributes or methods from multiple base classes. This can happen in multiple inheritance, where a class inherits from more than one base class, or in multilevel inheritance, where a class inherits from a base class that itself inherits from another base class.

Inheritance is a powerful feature in object-oriented programming (OOP) that allows one class to inherit properties and behavior from another class. However, it can also lead to ambiguity, which can make the code difficult to understand and maintain.

Ambiguity in Inheritance

Types

There are two types of ambiguity in inheritance:

- ✓ **Method Ambiguity**: This occurs when a derived class inherits multiple methods with the same name from different base classes.
- ✓ **Attribute Ambiguity**: This occurs when a derived class inherits multiple attributes with the same name from different base classes.

Ambiguity in Single Inheritance

```
#include <iostream>
using namespace std;
class Animal
{
  public:
   void eat()
   {
      cout << "The animal eats." << endl;
   }
};
class Mammal : public Animal
{
  public:
   void eat()
   {
      cout << "The mammal eats." << endl;
   }
};
```

```
int main()
{
   Mammal m;
   m.eat();
   return 0;
}
```

OUTPUT

```
The mammal eats.
```

By – Mohammad Imran

Ambiguity in Single Inheritance

Here in the given example there is no ambiguity but compiler will call **eat()** function of current class because object is created for the Mammal class.

So, the question is arises here how to call **eat()** function of base class.

Resolving Ambiguity

To resolve ambiguity in inheritance, we can use the following techniques:

- ✓ **Scope Resolution Operator:** We can use the scope resolution operator (::) to specify which base class's method or attribute to use.
- ✓ **Virtual Inheritance:** We can use virtual inheritance to ensure that a class inherits from only one instance of a base class, even if it inherits from multiple classes that themselves inherit from the same base class.
- ✓ **Interface-Based Inheritance:** We can use interface-based inheritance, where a class inherits from an interface rather than a base class, to avoid ambiguity.

Ambiguity in Single Inheritance

Example - 7

```
#include <iostream>
using namespace std;
class Animal
{
  public:
   void eat()
   {
   cout << "The animal eats." << endl;
   }
};
class Mammal : public Animal
{
  public:
   void eat()
   {
      cout << "The mammal eats." << endl;
   }
};
```

Ambiguity in Single Inheritance

Example - 7

```
class Dog : public Mammal
{
  public:
   void void eat()
   {
      cout << "The dog eats." << endl;
   }
};
int main()
{
   Dog myDog;
   myDog.eat();
   myDog.Animal::eat();
   myDog.Mammal::eat();
   return 0;
}
```

OUTPUT

```
The dog eats.
The animal eats.
The mammal eats.
```

By – Mohammad Imran

Ambiguity in Single Inheritance

Example - 7 **Explanation**

In this example, we have a single inheritance hierarchy: **Animal -> Mammal -> Dog**. Each class has an **eat()** method with a different implementation.

When we create a Dog object and call **myDog.eat(),** the compiler is not ambiguous because it will call the **eat()** method of the Dog class.

By – Mohammad Imran

Ambiguity in Single Inheritance

Example - 7 **Explanation**

However, if we want to call the **eat()** method of the Animal or Mammal class, we need to use the scope resolution operator **(::)** to specify which class's method to call. This is an example of ambiguity in single inheritance, where a derived class inherits conflicting methods from its base class.

To resolve this ambiguity, we can use the scope resolution operator to specify which class's method to call, as shown in the example.

Ambiguity in Multiple Inheritance

```
#include <iostream>
using namespace std;
class FlyingAnimal
{
  public:
   void walk()
   {
      cout << "The flying animal flies and walk." << endl;
   }
};
class WalkingAnimal
{
  public:
   void walk()
   {
      cout << "The walking animal walks." << endl;
   }
};
```

By – Mohammad Imran

Ambiguity in Multiple Inheritance

```
class Bird : public FlyingAnimal, public WalkingAnimal
{
  public:
   void sound()
   {
      cout << "The bird chirps." << endl;
   }
};
int main()
{
   Bird myBird;
   myBird.walk(); //Ambiguity for which method to call
   return 0;
}
```

By – Mohammad Imran

Ambiguity in Multiple Inheritance | Example - 8 | Resolution

```
#include <iostream>
using namespace std;
class FlyingAnimal
{
  public:
   void walk()
   {
      cout << "The flying animal flies and walk." << endl;
   }
};
class WalkingAnimal
{
  public:
   void walk()
   {
      cout << "The walking animal walks." << endl;
   }
};
```

By – Mohammad Imran

Ambiguity in Multiple Inheritance | Example - 8 | Resolution

```
class Bird : public FlyingAnimal, public WalkingAnimal
{
  public:
   void sound()
   {
      cout << "The bird chirps." << endl;
   }
};
int main()
{
   Bird brd;
   brd.sound();
   brd.FlyingAnimal::walk();
   brd.WalkingAnimal::walk();
   return 0;
}
```

OUTPUT

```
The bird chirps.
The flying animal flies and walk.
The walking animal walk.
```

By – Mohammad Imran

Ambiguity in Multiple Inheritance

```
#include <bits/stdc++.h>
using namespace std;
class Teacher
{
  public:
   void teach()
   {
      cout << "Teaching general subjects." << endl;
   }
};
class Musician
{
  public:
   void teach()
   {
      cout << "Teaching music." << endl;
   }
};
```

By – Mohammad Imran

Ambiguity in Multiple Inheritance

```
class MusicTeacher : public Teacher, public Musician
{
  public:
   void customTeach()
   {
      Teacher::teach();
   }
};

int main()
{
   MusicTeacher myTeacher;
   myTeacher.customTeach;
   return 0;
}
```

OUTPUT

```
Teaching general subject.
```

By – Mohammad Imran

Ambiguity in Multiple Inheritance

Syntax

Override The Member Function To Resolve Ambiguity Of Mulitiple Inheritance In C++

If we want to provide a custom implementation for the member in the derived class, we can override it by defining a newer version of the member function in the derived class. This kills all scope of ambiguity by providing a clear implementation in the derived class.

```
class   Derived   :   public   Base1,
public Base2

{

  public:

   void member_function_Name()

   {

      //Cunstom Implementation

   }

}
```

Ambiguity in Multiple Inheritance

Example - 9

Resolution

```
#include <bits/stdc++.h>
using namespace std;
class Teacher
{
  public:
   void teach()
   {
      cout << "Teaching general subjects." << endl;
   }
};
class Musician
{
  public:
   void teach()
   {
      cout << "Teaching music." << endl;
   }
};
```

By – Mohammad Imran

Ambiguity in Multiple Inheritance

Example - 9

Resolution

```
class MusicTeacher : public Teacher, public Musician
{
  public:
   void teach()
   {
      cout << "Teaching general and music subject both";
   }
};

int main()
{
   MusicTeacher mt;
   mt.Teach;
   return 0;
}
```

OUTPUT

```
Teaching general and
music subject both
```

By – Mohammad Imran

Diamond Problem in Multiple or Multipath Inheritance

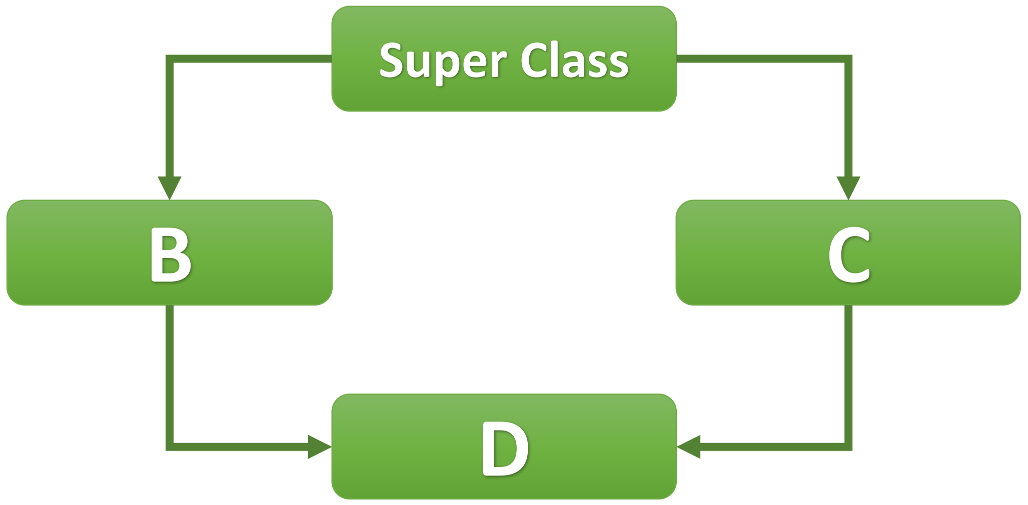
Multipath inheritance, also known as multiple inheritance, is a feature of some object-oriented programming languages that allows a class to inherit properties and behavior from more than one parent class. In C++, a class can inherit from multiple base classes using the : keyword.

However, multipath inheritance can lead to the Diamond Problem, which occurs when a class inherits from two classes that have a common base class. This creates an ambiguity in the inheritance hierarchy, making it unclear which path to follow when resolving method calls or accessing inherited members.

By – Mohammad Imran

Diamond Problem in Multiple or Multipath Inheritance

The Diamond Problem is one of the classic problems one faces when using multiple inheritance in C++. It arises when a class inherits from two or more classes that share a common base class. This common base class is inherited more than once through the chain of inheritance, creating ambiguity and potential conflicts in the derived class.



By – Mohammad Imran

Diamond Problem in Multiple or Multipath Inheritance

Let’s consider a real-world example to understand this better. Say we have a base class called Human and two derived classes, a Person representing individuals and an Employee class representing employees. Both Person and Employee classes inherit from a common base class, Human. Then we create a Manager class that inherits from both Person and Employee, and we encounter the Diamond pattern problem.

Diamond Problem in Multiple or Multipath Inheritance

Here's a step-by-step explanation of the Diamond Problem:

Base Class: Suppose you have a base class, let's call it Human, which contains a data member or function.

First Inheritance: We create two classes, say, Person and Employee, which both inherit from Human. Each of these classes may override or add their own members.

Second Inheritance: Now, we create a derived class, Manager, that inherits from both Person and Employee. This leads to the Manager class having two instances of the Human class, one through Person and another through Employee.

By – Mohammad Imran

Diamond Problem in Multiple or Multipath Inheritance

The ambiguity arises when we try to access or use members of the Human class within the Manager class. The compiler fails to determine which instance of the Human class to refer to, leading to ambiguity errors.

Note - This may sound like the ambiguity problem we discussed above, but there is a critical difference between the two. A diamond problem arises when we use multiple inheritance along with multi-level inheritance. However, ambiguity can still arise when employing just the multiple inheritance concept.

Diamond Problem

Ambiguity Example

```
#include <bits/stdc++.h>
using namespace std;
class Human
{
   private:
    string name;
   public:
    Human(const std::string &name) : name(name)
    {}
    void introduce()
    {
        cout << "I am a human named " << name << endl;
    }
};
```

By – Mohammad Imran

Diamond Problem

Ambiguity Example

```
class Person : public Human

{

   public:

    Person(const std::string &name) : Human(name)

    {}

    void greet()

    {

        cout << "Hello, I'm a person." << endl;

    }

};
```

By – Mohammad Imran

Diamond Problem

Ambiguity Example

```
class Employee : public Human
{
   public:
    Employee(const std::string &name) : Human(name)
    {}
    void work()
    {
        cout << "I am working as an employee." << endl;
    }
};
```

By – Mohammad Imran

Diamond Problem

Ambiguity Example

```
class Manager : public Person, public Employee
{
  public:
   Manager(const string &name) : Person(name), Employee(name)
   {}
};

int main()
{
   Manager mr("Shivani");
   mr.introduce(); // Ambiguity
   mr.greet();
   mr.work();
   return 0;
}
```

By – Mohammad Imran

Diamond Problem

Example - 10

Resolution of Diamond Problem Using Scope Resolution Operator

```
int main()
{
   string name;
   cout << "Enter name : ";
   cin >> name;
   Manager mr(name);
   mr.Person::introduce();
   mr.greet();
   mr.work();
   return 0;
}
```

OUTPUT

```
Enter name : Imran
I am a human named Imran
Hello, I'm a person.
I am working as an employee.
```

By – Mohammad Imran

Virtual Inheritance

Virtual inheritance is a feature in C++ that ensures that only one instance of a common base class is shared among multiple derived classes in a diamond-shaped inheritance hierarchy and eliminates the possibility of repetitive instances. When a base class is marked as virtual during inheritance, it signals the compiler to create a single instance of that base class within the hierarchy.

This single instance eliminates all possible ambiguity and conflicts, allowing us to access the shared base class without issues.

By – Mohammad Imran

Virtual Inheritance

How it Works

Virtual inheritance is a feature in C++ that allows a class to inherit virtually from a base class, ensuring that only one instance of the base class is created, even if multiple inheritance paths exist. This helps to resolve the Diamond Problem by avoiding ambiguity in the inheritance hierarchy.

When a class virtually inherits from a base class, it creates a virtual table (vtable) that contains a pointer to the base class's vtable. This allows the class to access the base class's members without creating multiple instances of the base class.

Virtual Inheritance

Diamond Problem

```
#include <iostream>
using namespace std;
class Animal
{
  public:
   void sound()
   {
      cout << "The animal makes a sound." << endl;
   }
};
class Mammal : public Animal
{
  public:
   void eat()     //sound()
   {
      cout << "The mammal eats." << endl;
   }
};
```

By – Mohammad Imran

Virtual Inheritance

Diamond Problem

```
class Bird : public Animal
{
  public:
   void fly() //sound()
   {
      cout << "The bird flies." << endl;
   }
};
class Bat : public Mammal, public Bird
{
  public:
   //sound(), sound(), eat(), fly()
   void useEcholocation()
   {
      cout << "The bat uses echolocation." << endl;
   }
};
```

By – Mohammad Imran

Virtual Inheritance

Diamond Problem

```
int main()
{
   Bat myBat;
   // Call methods from Animal, Mammal, and Bird
   // Output: The animal makes a sound. (via Mammal)
   myBat.sound();   // Ambiguity
   myBat.eat();     // Output: The mammal eats.
   myBat.fly();     // Output: The bird flies.
   // Output: The bat uses echolocation.
   myBat.useEcholocation();
   return 0;
}
```

By – Mohammad Imran

Virtual Inheritance | Diamond Problem Resolution | Example - 11

```
#include <iostream>
using namespace std;
class Animal
{
  public:
   void sound()
   {
      cout << "The animal makes a sound." << endl;
   }
};
class Mammal : virtual public Animal
{
  public:
   void eat()      //sound()
   {
      cout << "The mammal eats." << endl;
   }
};
```

By – Mohammad Imran

Virtual Inheritance

Diamond Problem

```
class Bird : virtual public Animal
{
  public:
   void fly() //sound()
   {
      cout << "The bird flies." << endl;
   }
};
class Bat : public Mammal, public Bird
{
  public:
   //eat(), fly(), Making virtual inheritance create only one copy of sound
   void useEcholocation()
   {
      cout << "The bat uses echolocation." << endl;
   }
};
```

By – Mohammad Imran

Virtual Inheritance

Diamond Problem

```
int main()
{
   Bat myBat;
   // Call methods from Animal, Mammal, and Bird
   // Output: The animal makes a sound. (via Mammal)
   myBat.sound();   // No Ambiguity Here
   myBat.eat();     // Output: The mammal eats.
   myBat.fly();     // Output: The bird flies.
   // Output: The bat uses echolocation.
   myBat.useEcholocation();
   return 0;
}
```

By – Mohammad Imran

Virtual Inheritance

Benefits

Virtual inheritance provides several benefits:

- ✓ **Avoids the Diamond Problem**: By ensuring that only one instance of the base class is created, virtual inheritance resolves the ambiguity in the inheritance hierarchy.
- ✓ **Reduces Memory Usage**: With virtual inheritance, only one instance of the base class is created, reducing memory usage and improving performance.
- ✓ **Improves Code Readability**: Virtual inheritance makes the inheritance hierarchy clearer and easier to understand, reducing the risk of errors and ambiguity.

By – Mohammad Imran

Defining Derived Class

Memory management is the process of controlling and coordinating a computer's main memory. It ensures that blocks of memory space are properly managed and allocated so the operating system (OS), applications and other running processes have the memory they need to carry out their operations.

Coding Questions

By – Mohammad Imran

Question - 1

Write a C++ program to demonstrate the ambiguity in multiple inheritance.

Click here to see code

By – Mohammad Imran

Question - 2

Write a C++ program to demonstrate the ambiguity resolution in multiple inheritance using the method overriding.

If we want to provide a custom implementation for the member in the derived class, we can override it by defining a newer version of the member function in the derived class. This kills all scope of ambiguity by providing a clear implementation in the derived class.

Click here to see code

By – Mohammad Imran

QUIZ

By – Mohammad Imran

Quiz - 1

```
class A
{
  public:
   void foo()
   {
      cout << "A::foo()" << endl;
   }
};
class B : public A
{
  public:
   void foo()
   {
      cout << "B::foo()" << endl;
   }
};
```

```
class C : public B
{
  public:
   void bar()
   {
      foo();
   }
};
int main()
{
   C obj;
   obj.bar();
   return 0;
}
```

OUTPUT

```
B::foo()
```

By – Mohammad Imran