FACULTY NAME:J.VIJAY SATHIARAJ
DESIGNAION:GUEST LECTURER
DEPARTMENT:GEOGRAPHY
CLASS:M.SC INTEGRATED
SEMESTER:V
SUBJECT:C++ PROGRAMMING
SUBJECT CODE:GE058

Unit: 4 Inheritance-single inheritance-multiple inheritance-multilevel-hybrid inheritance overloading.

Inheritance in C++

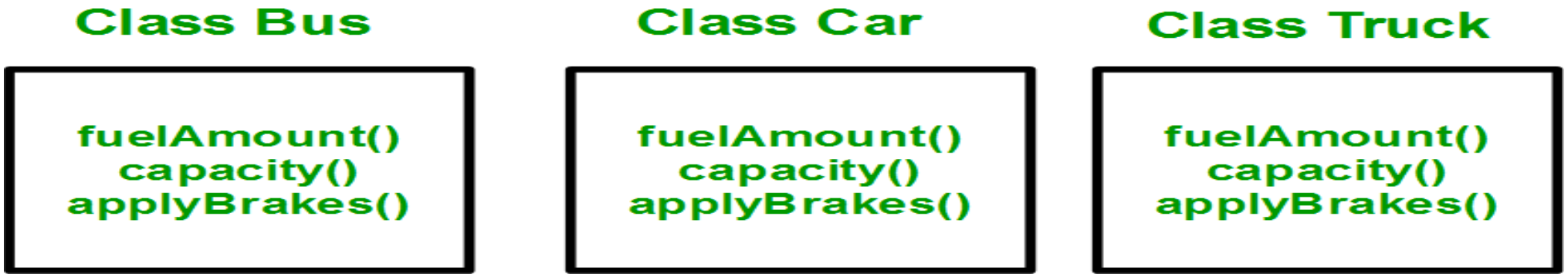
The capability of a class to derive properties and characteristics from another class is called **Inheritance**. Inheritance is one of the most important feature of Object Oriented Programming.

Sub Class: The class that inherits properties from another class is called Sub class or Derived Class.

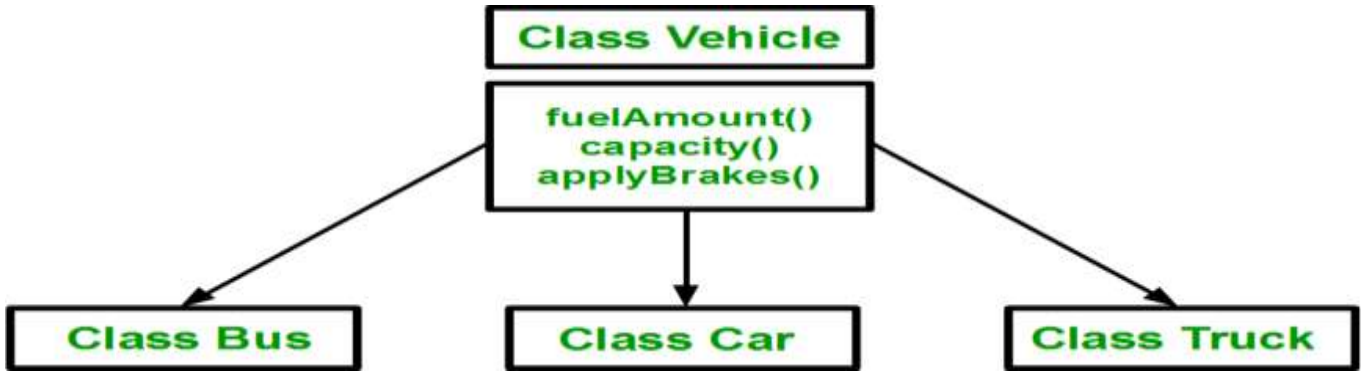
Super Class: The class whose properties are inherited by sub class is called Base Class or Super class.

Why and when to use inheritance?

Consider a group of vehicles. You need to create classes for Bus, Car and Truck. The methods fuelAmount(), capacity(), applyBrakes() will be same for all of the three classes. If we create these classes avoiding inheritance then we have to write all of these functions in each of the three classes as shown in below figure:



You can clearly see that above process results in duplication of same code 3 times. This increases the chances of error and data redundancy. To avoid this type of situation, inheritance is used. If we create a class Vehicle and write these three functions in it and inherit the rest of the classes from the vehicle class, then we can simply avoid the duplication of data and increase re-usability. Look at the below diagram in which the three classes are inherited from vehicle class:



Using inheritance, we have to write the functions only one time instead of three times as we have inherited rest of the three classes from base class(Vehicle).

Implementing inheritance in C++: For creating a sub-class which is inherited from the base class we have to follow the below syntax.

Syntax:

```
class subclass_name : access_mode base_class_name
{
 //body of subclass
};
```

Here, **subclass_name** is the name of the sub class, **access_mode** is the mode in which you want to inherit this sub class for example: public, private etc. and **base_class_name** is the name of the base class from which you want to inherit the sub class.

Modes of Inheritance

1. **Public mode**: If we derive a sub class from a public base class. Then the public member of the base class will become public in the derived class and protected members of the base class will become protected in derived class.
2. **Protected mode**: If we derive a sub class from a Protected base class. Then both public member and protected members of the base class will become protected in derived class.
3. **Private mode**: If we derive a sub class from a Private base class. Then both public member and protected members of the base class will become Private in derived class.

Note : The private members in the base class cannot be directly accessed in the derived class, while protected members can be directly accessed. For example, Classes B, C and D all contain the variables x, y and z in below example. It is just question of access.

The below table summarizes the above three modes and shows the access specifier of the members of base class in the sub class when derived in public, protected and private modes:

Base class member access specifier	Type of Inheritence		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)

Types of Inheritance in C++

Single Inheritance: In single inheritance, a class is allowed to inherit from only one class. i.e.one sub class is inherited by one base class only

Syntax:

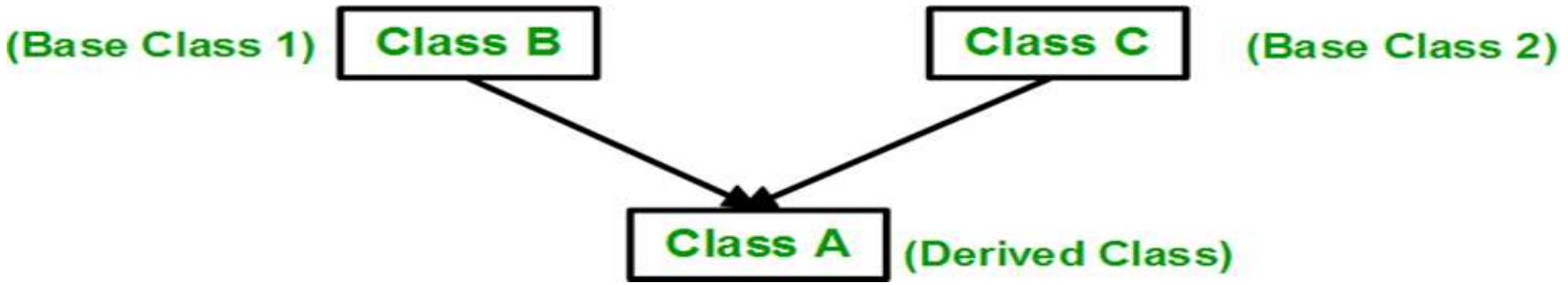
```
class subclass_name : access_mode base_class
{
//body of subclass
};
```

```
// C++ program to explain
// Single inheritance #include <iostream> using
namespace std;

// base class class Vehicle { public:
  Vehicle()
  {
   cout << "This is a Vehicle" << endl;
  }
};
```

```
    // sub class derived from two base classes
    class Car: public Vehicle{
};
// main function int main()
    {
      // creating object of sub class will
      // invoke the constructor of base classes Car
      obj;
      return 0;}
      Output:
This is a vehicle
```

Multiple Inheritance: Multiple Inheritance is a feature of C++ where a class can inherit from more than one classes. i.e one **sub class** is inherited from more than one **base class**




```
class subclass_name : access_mode base_class1, access_mode
 base_class2, ....
{
 //body of subclass
};
```

Here, the number of base classes will be separated by a comma (‘, ‘) and access mode for every base class must be specified.

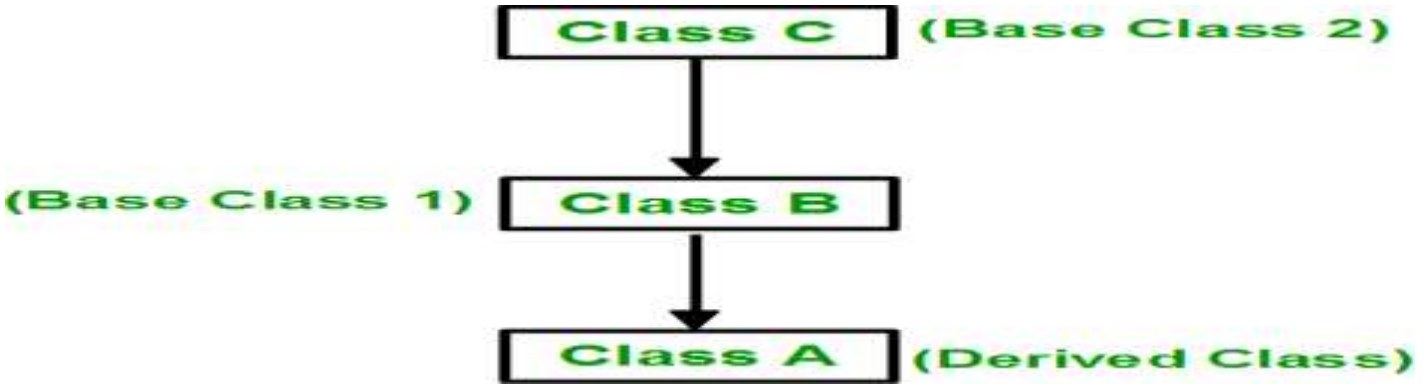
```
// C++ program to explain
    // multiple inheritance #include <iostream> using namespace
    std;
// first base class class Vehicle { public:
      Vehicle()
```

```
        {
         cout << "This is a Vehicle" << endl;
        }
      };
// second base class class Four Wheeler { public:
        Four Wheeler()
        {
         cout << "This is a 4 wheeler Vehicle" << endl;
        }
      };
// sub class derived from two base classes
     class Car: public Vehicle, public Four Wheeler {
};
// main function int main()
      {
        // creating object of sub class will
        // invoke the constructor of base classes Car obj;
        return 0;
      }
```

Output:

This is a Vehicle
This is a 4 wheeler Vehicle

Multilevel Inheritance: In this type of inheritance, a derived class is created from another derived class.




```
// C++ program to implement
// Multilevel Inheritance
 #include <iostream>
 using namespace std;
```

```
// base class class Vehicle
{
public: Vehicle()
{
cout << "This is a Vehicle" << endl;
}
};
class fourWheeler: public Vehicle
{ public: fourWheeler()
{
cout<<"Objects with 4 wheels are vehicles"<<endl;
}
};
// sub class derived from two base classes class Car: public fourWheeler{
public: car()
{
cout<<"Car has 4 Wheels"<<endl;
}
```

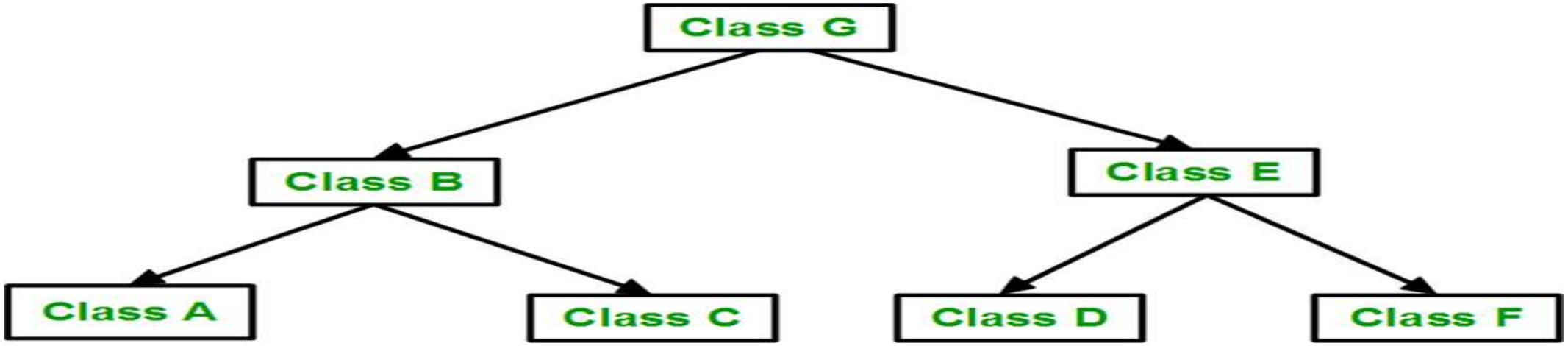
```
};

// main function
 int main()
{
 //creating object of sub class will
//invoke the constructor of base classes Car obj;
return 0
 }
```

OUTPUT:

```
This is a Vehicle
Objects with 4 wheels are vehicles Car has 4 Wheels
```

4.Hierarchical Inheritance: In this type of inheritance, more than one sub class is inherited from a single base class. i.e. more than one derived class is created from a single base class.




```
// C++ program to implement
// Hierarchical Inheritance #include <iostream> using namespace std;

// base class class Vehicle
{
public: Vehicle()
{
cout << "This is a Vehicle" << endl;
}
};
```

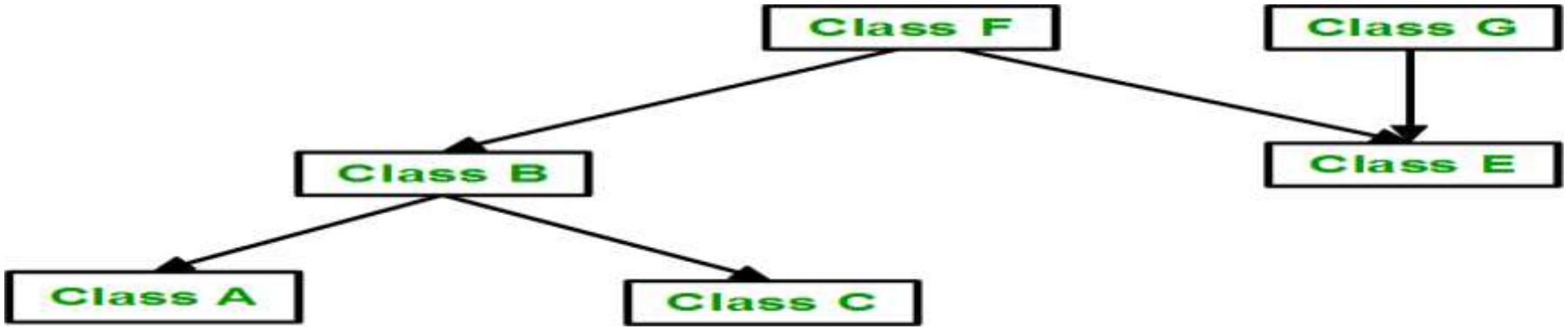
```
// first sub class
 class Car: public Vehicle
 {
 };
 // second sub class
 class Bus: public Vehicle
 {
 };
 // main function
  int main()
 {
 // creating object of sub class will
 // invoke the constructor of base class Car obj1;
 Bus obj2;
 return 0;
 }
```

OUTPUT: This is a Vehicle

This is a Vehicle

•**Hybrid (Virtual) Inheritance**: Hybrid Inheritance is implemented by combining more than one type of inheritance. For example: Combining Hierarchical inheritance and Multiple Inheritance.

Below image shows the combination of hierarchical and multiple inheritance:




```
// C++ program for Hybrid Inheritance
 #include <iostream>
using namespace std;
  // base class
   class Vehicle
  {
```

```
public: Vehicle()
{
cout << "This is a Vehicle" << endl;
}
};
 //base class class Fare
{
public: Fare()
{
cout<<"Fare of Vehicle\n";
}
};
 // first sub class
class Car: public Vehicle
{
 };
```

```
 // second sub class
class Bus: public Vehicle, public Fare
{

};
// main function int main()
{
// creating object of sub class will
// invoke the constructor of base class Bus obj2;
return 0;
}
```

OUPUT:

```
This is a Vehicle
Fare of Vehicle
```

C++ Overloading (Function and Operator)

If we create two or more members having the same name but different in number or type of parameter, it is known as C++ overloading. In C++, we can overload:

-methods,

- constructors, and

- indexed properties

It is because these members have parameters only.

Types of overloading in C++ are:

- Function overloading

- Operator overloading

C++ Function Overloading

Function Overloading is defined as the process of having two or more function with the same name, but different in parameters is known as function overloading in C++. In function overloading, the function is redefined by using either different types of arguments or a different number of arguments. It is only through these differences compiler can differentiate between the functions.

The advantage of Function overloading is that it increases the readability of the program because you don't need to use different names for the same action.

C++ Function Overloading Example

Let's see the simple example of function overloading where we are changing number of arguments of add() method.

// program of function overloading when number of arguments vary.

```
1.	#include <iostream>
2.	using namespace std;
3.	class Cal {
```

```
4.      public:
5.      static int add(int a,int b){
6.      return a + b;
7.      }
8. static int add(int a, int b, int c)
9.      {
10.     return a + b + c;
11.     }
12. };
13. int main(void) {
14.     Cal C;  //          class object declaration.
15.     cout<<C.add(10, 20)<<endl;
16.     cout<<C.add(12, 20, 23);
17.     return 0;
18. }
```

Output:

```
30
55
```

C++ Operators Overloading

Operator overloading is a compile-time polymorphism in which the operator is overloaded to provide the special meaning to the user-defined data type. Operator overloading is used to overload or redefines most of the operators available in C++. It is used to perform the operation on the user-defined data type. For example, C++ provides the ability to add the variables of the user-defined data type that is applied to the built-in data types.

The advantage of Operators overloading is to perform different operations on the same operand.

Operator that cannot be overloaded are as follows:

1)Scope operator (::)
2)Size of
3)member selector(.)
4)member pointer selector(*)
5)ternary operator(?:)

Syntax of Operator Overloading

```
1.      return_type class_name : : operator op(argument_list)
2.      {
3.      // body of the function.
4. }
```

Where the return type is the type of value returned by the function.

class_name is the name of the class.

operator op is an operator function where op is the operator being overloaded, and the operator is the keyword.

Rules for Operator Overloading

1) Existing operators can only be overloaded, but the new operators cannot be overloaded.
2) The overloaded operator contains atleast one operand of the user-defined data type.
3) We cannot use friend function to overload certain operators. However, the member function can be used to overload those operators.
4) When unary operators are overloaded through a member function take no explicit arguments, but, if they are overloaded by a friend function, takes one argument.
5) When binary operators are overloaded through a member function takes one explicit argument, and if they are overloaded through a friend function takes two explicit arguments.

C++ Operators Overloading Example

Let's see the simple example of operator overloading in C++. In this example, void operator ++ () operator function is defined (inside Test class).

```
// program to overload the unary operator ++.
1.      #include <iostream>
2.      using namespace std;
3.      class Test
4.      {
5.      private:
6.      int num;
7.      public:
8.      Test(): num(8){}
9.      void operator ++()      {
10.     num = num+2;
11.     }
12.     void Print() {
13.     cout<<"The Count is: "<<num;
14.     }
```

```
15. };
16.       int main()
17.       {
18.       Test tt;
19.       ++tt; // calling of a function "void operator ++()"
20.       tt.Print();
21.       return 0;
22. }
```

Output:

```
The Count is: 10
```