SCS1310: Object-Oriented Modelling and Programming

Inheritance

Viraj Welgama

Focus on the Course Title…

Procedural Programming vs OO Programing…

	OOP	PP
Design around	Data & Objects	Actions & logic
What and how	How to do is embedded inside object and triggered by what to do. "Message passing" is a form of communication between objects	No such separation. A series of logical procedures process and pass on the data to produce output.
Advantages	Intuitive Data security Robustness Flexibility Extensibility Maintainability	If speed of the execution is very critical, Memory constraints are very high (like in Microcontroller, device drivers etc.) Code is small, Changes are not frequent, Procedural code recommended.

Supporting Concepts of Object Orientation

1. Objects:
   - Instances of classes that contain data and methods that operate on that data.
2. Classes:
   - A blueprint or template for creating objects, defines the attributes and behavior of objects.
3. Association & Aggregation:
   - Aggregation is the relationship between objects where an object is composed of one or more objects (*has-a*)
   - Association is the General relationship between two classes.
4. Composition:
   - Combining objects of different classes to build more complex functionality.

Supporting Concepts of Object Orientation

5. Access Specifiers:
   - Control the visibility of class members (*private, protected, public*).
6. Constructors and Destructors:
   - Special methods invoked when an object is created and destroyed.
   - Reproduction capability of a class.

Core Principles of Object Orientation

1. Inheritance:
   - A mechanism that allows one class to inherit properties and behaviors from a parent class.
2. Polymorphism:
   - The ability of objects to take on multiple forms. This allows objects of different classes to be used interchangeably.
3. Encapsulation:
   - The idea of wrapping data and functions that operate on that data within a single unit, or object. This is used to keep the data safe from outside interference and misuse.
4. Abstraction:
   - Hiding the implementation details and showing only the necessary information.

Inheritance

Inheritance

- The child is satisfied with this answer because she can take all of her knowledge about horses and apply it to zebras, with the additional fact that zebras have stripes.
- We do essentially the same thing when we use inheritance in our code.

In OOP: Inheritance

- The capability of a class to derive properties and characteristics from another.
- Inheritance is one of the most important feature of Object Oriented Programming.
- Inheritance is also called an *is-a* hierarchy.
- Inheritance is the key for code reuse.
  - If a base class has been created, then a derived class can be created to add more information

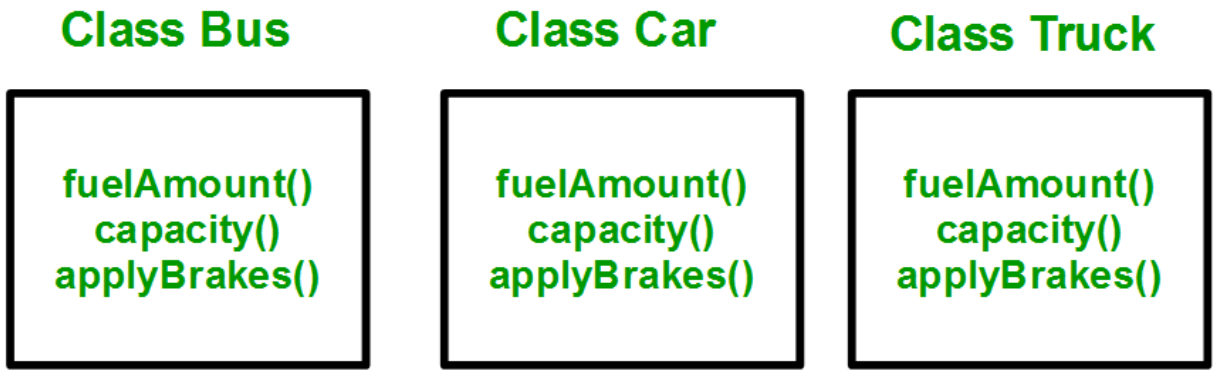
Inheritance: Terminology

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

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

Example:

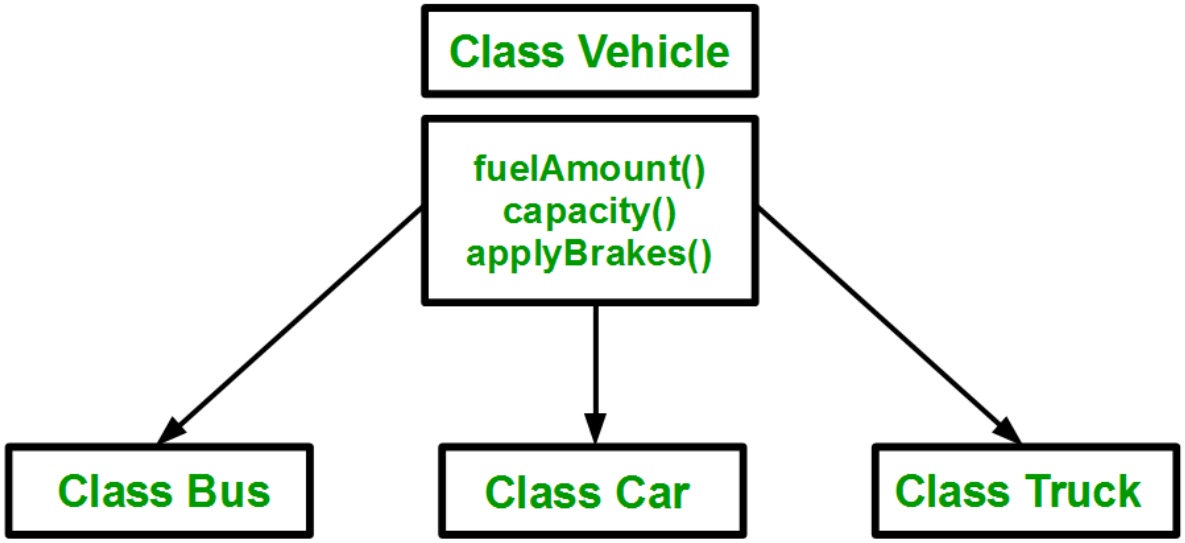
- Consider a group of vehicles.
- You need to create classes for Bus, Car and Truck.
- The methods fuelAmount(), capacity(), applyBrakes() will be same for all of the three classes.
- If we create these classes avoiding inheritance then we have to write all of these functions in each of the three classes.



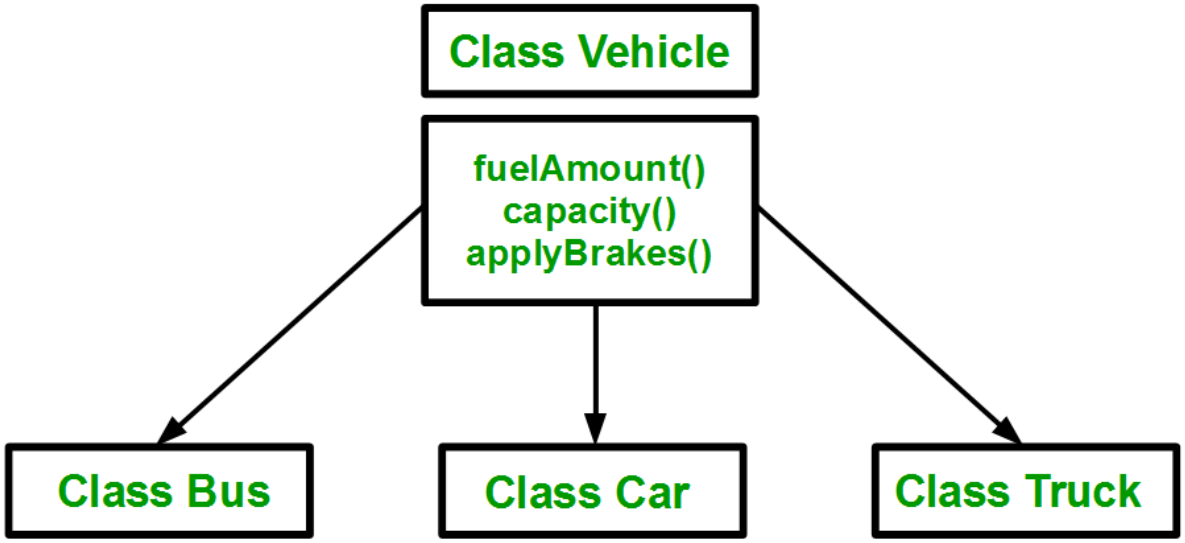
Example: Issues

- Increase the chances of errors
- Data redundancy

Example: Solution



Example: Solution



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.

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

Implementing Inheritance in C++

- For creating a sub-class which is inherited from the base class, we have to follow the below syntax.

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

- **subclass_name** is the name of the sub class
- **access_mode** is the mode in which you want to inherit this sub class (public, private etc.)
- **base_class_name** is the name of the base class from which you want to inherit the sub class

Example:

```
//Base class
class Parent
{
    public:
      int id_p;
};

// Sub class inheriting from Base Class(Parent)
class Child : public Parent
{
    public:
      int id_c;
};

//main function
int main()
   {

       Child obj1;

       // An object of class child has all data members
       // and member functions of class parent
       obj1.id_c = 7;
       obj1.id_p = 91;
       cout << "Child id is " <<  obj1.id_c << endl;
       cout << "Parent id is " <<  obj1.id_p << endl;

       return 0;
   }
```

Example:

the 'Child' class is publicly inherited from the 'Parent' class so the public data members of the class 'Parent' will also be inherited by the class 'Child'.

```
//Base class
class Parent
{
    public:
      int id_p;
};

// Sub class inheriting from Base Class(Parent)
class Child : public Parent
{
    public:
      int id_c;
};

//main function
int main()
   {

       Child obj1;

       // An object of class child has all data members
       // and member functions of class parent
       obj1.id_c = 7;
       obj1.id_p = 91;
       cout << "Child id is " <<  obj1.id_c << endl;
       cout << "Parent id is " <<  obj1.id_p << endl;

       return 0;
   }
```

Example:

the 'Child' class is publicly inherited from the 'Parent' class so the public data members of the class 'Parent' will also be inherited by the class 'Child'.

```
Child id is 7
Parent id is 91
```

```
//Base class
class Parent
{
    public:
      int id_p;
};

// Sub class inheriting from Base Class(Parent)
class Child : public Parent
{
    public:
      int id_c;
};

//main function
int main()
   {

       Child obj1;

       // An object of class child has all data members
       // and member functions of class parent
       obj1.id_c = 7;
       obj1.id_p = 91;
       cout << "Child id is " <<  obj1.id_c << endl;
       cout << "Parent id is " <<  obj1.id_p << endl;

       return 0;
   }
```

Modes of Inheritance

- **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.
- **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.
- **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.

Modes of Inheritance

- The private members in the base class cannot be directly accessed in the derived class, while protected members can be directly accessed.
- Classes B, C and D in bellow example, contain the variables x, y and z, but it is just question of access.

Modes of Inheritance: Example

```
// C++ Implementation to show that a derived class
// doesn't inherit access to private data members.
// However, it does inherit a full parent object

class A
{
public:
    int x;
protected:
    int y;
private:
    int z;
};

class B : public A
{
    // x is public
    // y is protected
    // z is not accessible from B
};

class C : protected A
{
    // x is protected
    // y is protected
    // z is not accessible from C
};

class D : private A    // 'private' is default for classes
{
    // x is private
    // y is private
    // z is not accessible from D
};
```

Access Mode: Summary

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)

Inheriting Constructors…

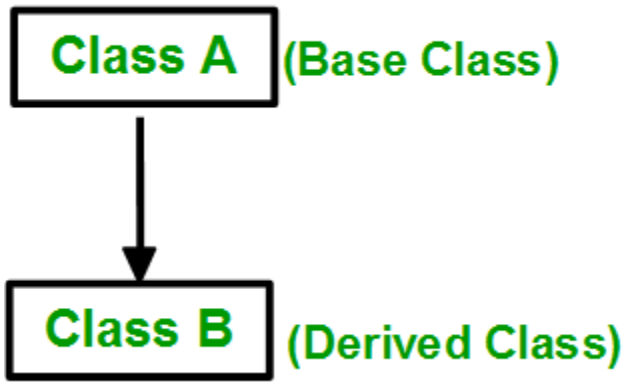
- A derived class will **NOT** inherit the constructors, destructor or assignment operator from a base class.
- Derived class constructors and assignment operators, however, **can call** base class constructors and assignment operators.
- The fact that these functions are not inherited does not mean they do not exist for the derived class.
- The compiler will automatically define a constructor, a copy constructor, a destructor and an assignment operator for a class that does not explicitly define these functions.
- When an instance of a derived class comes into existence, both the base class constructor and the derived class constructor are automatically invoked.

Inheriting Constructors…

- The base class constructor initializes the data members inherited from the base class, and the derived class constructor initializes the data members declared in the derived class definition.
- The constructors are invoked in the order of derivation. ***Base*** then ***derived***.
- If more than one version of the base class constructor exists, the default constructor will be invoked automatically. To have any other constructor invoked, you must use the constructor initialization list.
- If a derived class does not explicitly define the copy constructor, the compiler will automatically provide a copy constructor that invokes the base class copy constructor, and makes a bitwise copy of any data members declared for the derived class.

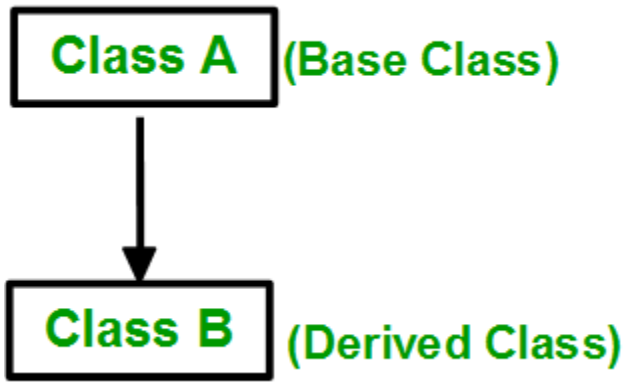
Single Inheritance

- 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.



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_name
{
  //body of subclass
};
```

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{
public:
    Car()
    {
      cout << "This is a car" << endl;
    }

};

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

Single Inheritance

```
This is a Vehicle
This is a car
```

```cpp
#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{
public:
    Car()
    {
      cout << "This is a car" << endl;
    }

};

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

Example: Chef

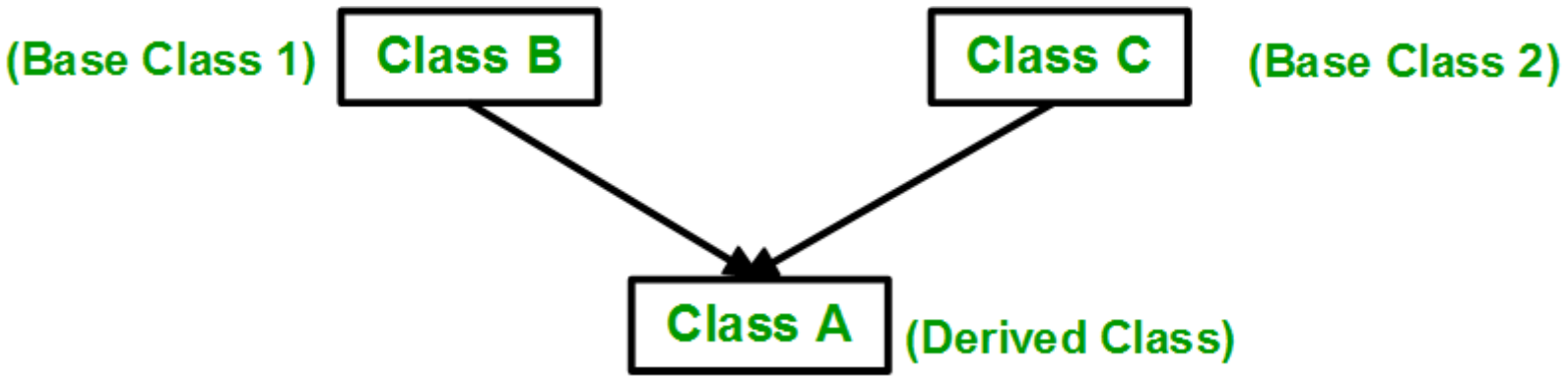
```
using namespace std;
class Chef{

    public:
        int age;
        void makeChicken(){
            cout << "The chef makes chicken" << endl;
        }

        void makeSalad(){
            cout << "The chef makes salad" << endl;
        }

        void makeSpecialDish(){
            cout << "The chef makes a special dish" << endl;
        }
        Chef(){
          cout<<"hi.. i am chef..."<<endl;
        }
};
class ItalianChef : public Chef{
    public:
        void makePasta(){
            cout << "The chef makes pasta" << endl;
        }
        // override
        void makeSpecialDish(){
            cout << "The chef makes chicken parm" << endl;
        }
        ItalianChef(){
          cout<<"hi, i am a chef born in Italy..."<<endl;
        }
};
int main(){

    Chef myChef;
    myChef.makeChicken();

    ItalianChef myItalianChef;
    myItalianChef.makeChicken();
        myChef.age = 30;
    myChef.makeSpecialDish();
    myItalianChef.makeSpecialDish();
    return 0;
}
```

Multiple Inheritance

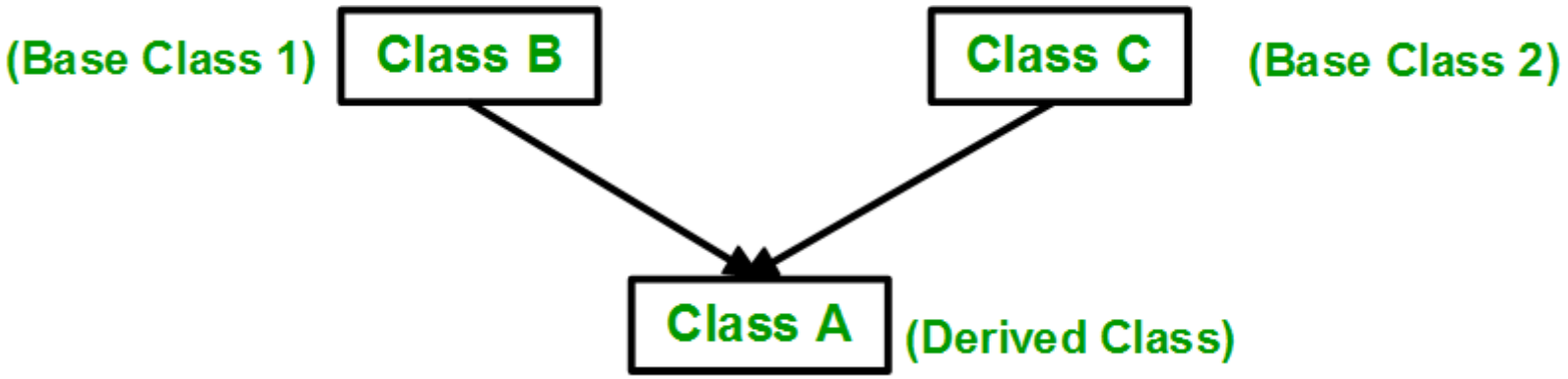
- Multiple Inheritance is a feature of C++ where a class can inherit from more than one classes.
  - i.e one sub class is inherited from more than one base classes.

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 classes.



Multiple Inheritance



Syntax:

```
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.

Multiple Inheritance

- The constructors of inherited classes are called in the same order in which they are inherited.
- For example, in the following program, Vehicle's constructor is called before FourWheeler's constructor.
- However, The destructors are called in reverse order of constructors.

Multiple Inheritance: Example

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

// second base class
class FourWheeler {
  public:
    FourWheeler() {
      cout << "This is a 4 wheeler Vehicle" << endl;
    }
};

// sub class derived from two base classes
class Car: public Vehicle, public FourWheeler {

};

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

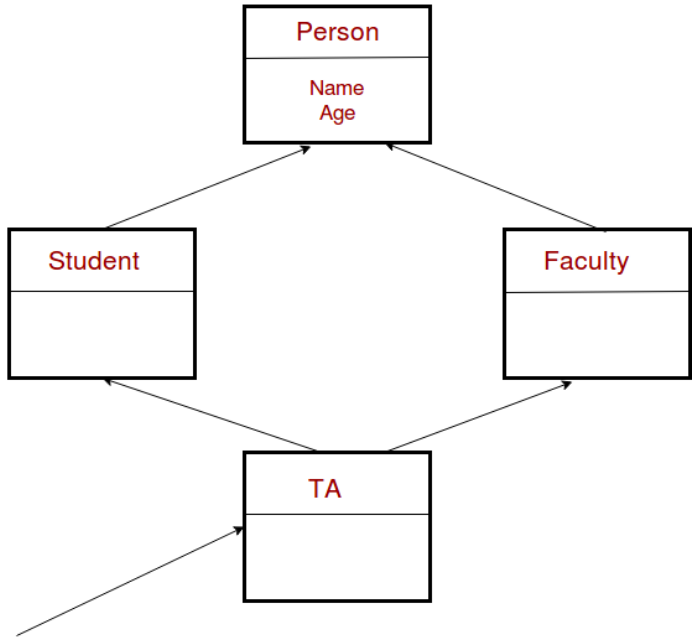
Multiple Inheritance: Example

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

// second base class
class FourWheeler {
  public:
    FourWheeler() {
      cout << "This is a 4 wheeler Vehicle" << endl;
    }
};

// sub class derived from two base classes
class Car: public Vehicle, public FourWheeler {

};

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

```
This is a Vehicle
This is a 4 wheeler Vehicle
```

The Diamond Problem

- The diamond problem occurs when two superclasses of a class have a common base class.
- For example, in the following diagram, the TA class gets two copies of all attributes of Person class, this causes ambiguities.

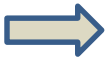


The Diamond Problem: Example

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

class Person {
   // Data members of person
public:
    Person(int x) { cout << "Person::Person(int ) called" << endl; }
};

class Faculty : public Person {
   // data members of Faculty
public:
    Faculty(int x):Person(x) {
       cout<<"Faculty::Faculty(int ) called"<< endl; }
};

class Student : public Person {
   // data members of Student
public:
    Student(int x):Person(x) {
        cout<<"Student::Student(int ) called"<< endl; }
};

class TA : public Faculty, public Student {
public:
    TA(int x):Student(x), Faculty(x) {
        cout<<"TA::TA(int ) called"<< endl; }
};

int main() {
    TA ta1(30);
}
```

The Diamond Problem: Example

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

class Person {
   // Data members of person
public:
    Person(int x) { cout << "Person::Person(int ) called" << endl; }
};

class Faculty : public Person {
   // data members of Faculty
public:
    Faculty(int x):Person(x) {
       cout<<"Faculty::Faculty(int ) called"<< endl; }
};

class Student : public Person {
   // data members of Student
public:
    Student(int x):Person(x) {
        cout<<"Student::Student(int ) called"<< endl; }
};

class TA : public Faculty, public Student {
public:
    TA(int x):Student(x), Faculty(x) {
        cout<<"TA::TA(int ) called"<< endl; }
};

int main() {
    TA ta1(30);
}
```



```
Person::Person(int ) called
Faculty::Faculty(int ) called
Person::Person(int ) called
Student::Student(int ) called
TA::TA(int ) called
```

The Diamond Problem: Example

- In the above program, constructor of ‘Person’ is called two times.
- Destructor of ‘Person’ will also be called two times when object ‘ta1’ is destructed.
- So object ‘ta1’ has two copies of all members of ‘Person’, this causes ambiguities.

'*virtual*' Inheritance

- One of the solution to this diamond problem is to *virtually* inherit from the base classes.
- We make the 'Faculty' and 'Student' classes by *virtually* inherit from the 'Person' base class to avoid two copies of 'Person' in 'TA' class.
- Then 'Person' is referred to as a *virtual base class* because it is inherited virtually.
- you have to use 'virtual' keyword to *virtually* inherit from the base classes.

Example:

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

class Person {
   // Data members of person
public:
    Person() { cout << "Person::Person() called" << endl; }
    Person(int x) { cout << "Person::Person(int ) called" << endl; }
};

class Faculty : virtual public Person {
   // data members of Faculty
public:
    Faculty(int x):Person(x) {
       cout<<"Faculty::Faculty(int ) called"<< endl; }
};

class Student : virtual public Person {
   // data members of Student
public:
    Student(int x):Person(x) {
        cout<<"Student::Student(int ) called"<< endl; }
};

class TA : public Faculty, public Student {
public:
    TA(int x):Student(x), Faculty(x) {
        cout<<"TA::TA(int ) called"<< endl; }
};

int main() {
    TA ta1(30);
}
```

Example:

```
Person::Person() called
Faculty::Faculty(int ) called
Student::Student(int ) called
TA::TA(int ) called
```

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

class Person {
   // Data members of person
public:
    Person() { cout << "Person::Person() called" << endl; }
    Person(int x) { cout << "Person::Person(int ) called" << endl; }
};

class Faculty : virtual public Person {
   // data members of Faculty
public:
    Faculty(int x):Person(x) {
       cout<<"Faculty::Faculty(int ) called"<< endl; }
};

class Student : virtual public Person {
   // data members of Student
public:
    Student(int x):Person(x) {
        cout<<"Student::Student(int ) called"<< endl; }
};

class TA : public Faculty, public Student {
public:
    TA(int x):Student(x), Faculty(x) {
        cout<<"TA::TA(int ) called"<< endl; }
};

int main() {
    TA ta1(30);
}
```

Example:

- In the above program, constructor of ‘Person’ is called once.
- One important thing to note in the above output is, the *default constructor* of ‘Person’ is called.
- When we use ‘virtual’ keyword, the default constructor of grandparent class is called by default even if the parent classes explicitly call parameterized constructor.

How to call the parameterized constructor of the ‘Person’ class?

- The constructor has to be called in ‘TA’ class.

```
class TA : public Faculty, public Student {
public:
    TA(int x):Student(x), Faculty(x), Person(x) {
        cout<<"TA::TA(int ) called"<< endl; }
};
```

How to call the parameterized constructor of the 'Person' class?

- The constructor has to be called in 'TA' class.

```
class TA : public Faculty, public Student {
public:
    TA(int x):Student(x), Faculty(x), Person(x) {
        cout<<"TA::TA(int ) called"<< endl; }
};
```

- In general, it is not allowed to call the grandparent's constructor directly, it has to be called through parent class.
- It is allowed only when 'virtual' keyword is used.

Avoiding ambiguity using scope resolution operator:

- Using scope resolution operator, we can manually specify the path from which data member a will be accessed

Avoiding ambiguity using scope resolution operator:

- Using scope resolution operator, we can manually specify the path from which data member a will be accessed

```
class ClassA {
    public:
        int a;
};

class ClassB: public ClassA {
    public:
        int b;
};

class ClassC: public ClassA {
    public:
        int c;
};

class ClassD: public ClassB, public ClassC {
    public:
        int d;
};

int main() {
    ClassD obj;
    //obj.a = 10; //compiler error: request for member 'a' is ambiguous
    obj.ClassB::a = 10;
    obj.ClassC::a = 100;
    obj.b = 20;
    obj.c = 30;
    obj.d = 40;
    cout<<"\n A in B: "<<obj.ClassB::a;
    cout<<"\n A in C: "<<obj.ClassC::a;
    cout<<"\n B: "<<obj.b;
    cout<<"\n C: "<<obj.c;
    cout<<"\n D: "<<obj.d;
}
```

Avoiding ambiguity using scope resolution operator:

- Using scope resolution operator, we can manually specify the path from which data member a will be accessed

```
A in B: 10
A in C: 100
B: 20
C: 30
D: 40
```

```
class ClassA {
    public:
        int a;
};

class ClassB: public ClassA {
    public:
        int b;
};

class ClassC: public ClassA {
    public:
        int c;
};

class ClassD: public ClassB, public ClassC {
    public:
        int d;
};

int main() {
    ClassD obj;
    //obj.a = 10; //compiler error: request for member 'a' is ambiguous
    obj.ClassB::a = 10;
    obj.ClassC::a = 100;
    obj.b = 20;
    obj.c = 30;
    obj.d = 40;
    cout<<"\n A in B: "<<obj.ClassB::a;
    cout<<"\n A in C: "<<obj.ClassC::a;
    cout<<"\n B: "<<obj.b;
    cout<<"\n C: "<<obj.c;
    cout<<"\n D: "<<obj.d;
}
```

Solving the Ambiguity in the Diamond Problem: Summary

- In the above example, both ClassB & ClassC inherit ClassA, they both have single copy of ClassA.
- However ClassD inherit both ClassB & ClassC, therefore ClassD have two copies of ClassA, one from ClassB and another from ClassC.
- If we need to access the data member a of ClassA through the object of ClassD, we must specify the path from which a will be accessed, whether it is from ClassB or ClassC, bco'z compiler can't differentiate between two copies of ClassA in ClassD.

- There are 2 ways to avoid this ambiguity:
  1. Use virtual base class
  2. Use scope resolution operator

Solving the Ambiguity

1. Use virtual base class

```
class ClassA {
    public:
        int a;
};

class ClassB: virtual public ClassA {
    public:
        int b;
};

class ClassC: virtual public ClassA {
    public:
        int c;
};

class ClassD: public ClassB, public ClassC {
    public:
        int d;
};

int main() {
    ClassD obj;
    obj.a = 10;
    obj.a = 100;
    obj.b = 20;
    obj.c = 30;
    obj.d = 40;
    cout<<"\n A in B: "<<obj.a;
    cout<<"\n A in C: "<<obj.a;
    cout<<"\n B: "<<obj.b;
    cout<<"\n C: "<<obj.c;
    cout<<"\n D: "<<obj.d;
}
```

Solving the Ambiguity

1. Use virtual base class

```
A in B: 100
A in C: 100
B: 20
C: 30
D: 40
```

```
class ClassA {
    public:
        int a;
};

class ClassB: virtual public ClassA {
    public:
        int b;
};

class ClassC: virtual public ClassA {
    public:
        int c;
};

class ClassD: public ClassB, public ClassC {
    public:
        int d;
};

int main() {
    ClassD obj;
    obj.a = 10;
    obj.a = 100;
    obj.b = 20;
    obj.c = 30;
    obj.d = 40;
    cout<<"\n A in B: "<<obj.a;
    cout<<"\n A in C: "<<obj.a;
    cout<<"\n B: "<<obj.b;
    cout<<"\n C: "<<obj.c;
    cout<<"\n D: "<<obj.d;
}
```

Solving the Ambiguity

1. Use virtual base class

```
A in B: 100
A in C: 100
B: 20
C: 30
D: 40
```

According to the above example, ClassD has only one copy of ClassA, therefore, 3rd statement (in main()) will overwrite the value of a, given at 2nd statement.

```
class ClassA {
    public:
        int a;
};

class ClassB: virtual public ClassA {
    public:
        int b;
};

class ClassC: virtual public ClassA {
    public:
        int c;
};

class ClassD: public ClassB, public ClassC {
    public:
        int d;
};

int main() {
    ClassD obj;
    obj.a = 10;
    obj.a = 100;
    obj.b = 20;
    obj.c = 30;
    obj.d = 40;
    cout<<"\n A in B: "<<obj.a;
    cout<<"\n A in C: "<<obj.a;
    cout<<"\n B: "<<obj.b;
    cout<<"\n C: "<<obj.c;
    cout<<"\n D: "<<obj.d;
}
```

Exercise:

What is the output of this program?

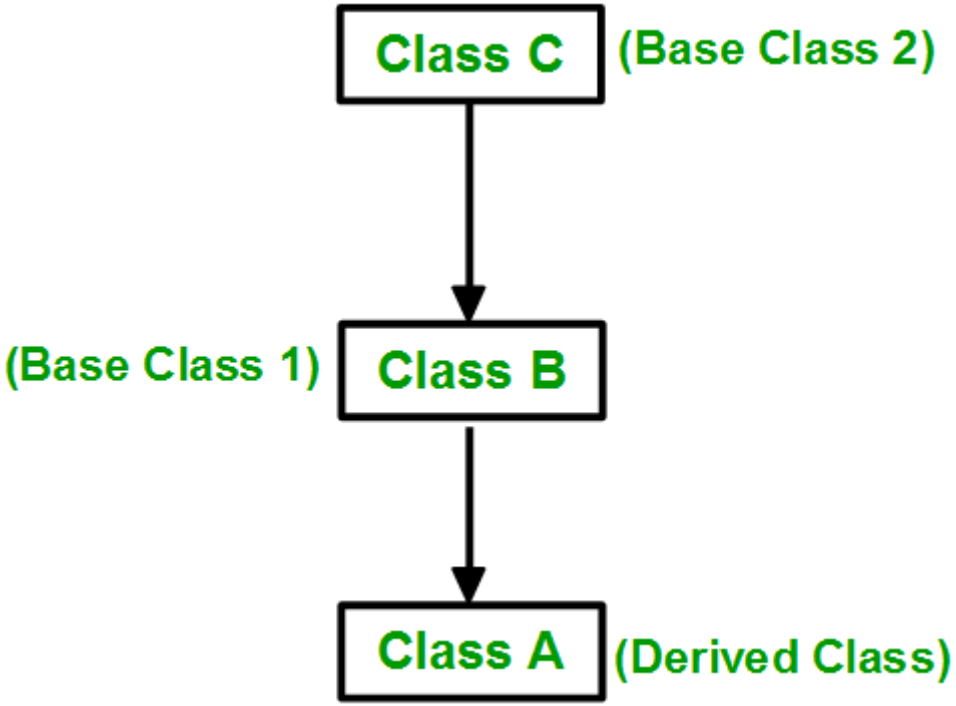
```
#include<iostream>
using namespace std;

class A {
  int x;
public:
  void setX(int i) {x = i;}
  void print() { cout << x; }
};

class B:  public A {
public:
  B()  { setX(10); }
};

class C:  public A {
public:
  C()  { setX(20); }
};

class D: public B, public C {
};

int main() {
    D d;
    d.print();
    return 0;
}
```

Multilevel Inheritance

- In this type of inheritance, a derived class is created from another derived class.

Multilevel Inheritance

- In this type of inheritance, a derived class is created from another derived class.



Multilevel Inheritance: Example

```
// 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;
}
```

Multilevel Inheritance: Example

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

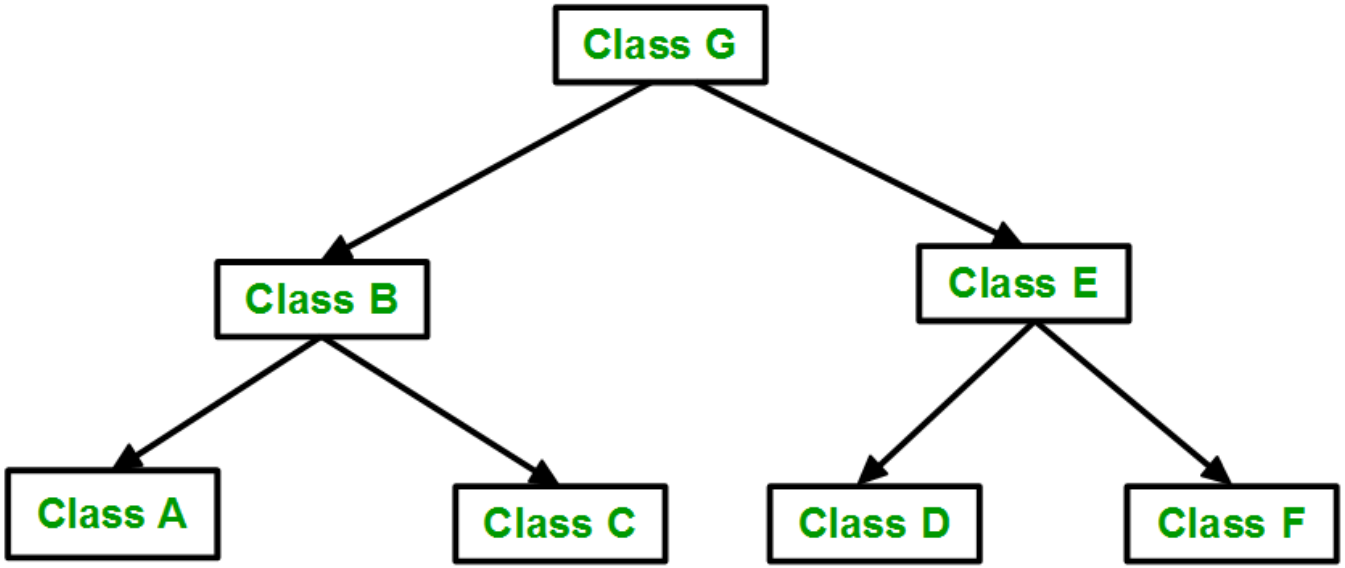
```cpp
// base class
class Vehicle {
  public:
    Vehicle()
    {
      cout << "This is a Vehicle" << endl;
    }
};

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;
}
```

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.

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.



Hierarchical Inheritance: Example

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

class Bus: public Vehicle {
  public:
    Bus()
    {
      cout<<"This is a bus..."<<endl;
    }
};

// sub class derived from two base classes
class Car: public Vehicle {
   public:
     Car()
     {
       cout<<"This is a car..."<<endl;
     }
};

// main function
int main() {
    Bus obj1;
    Car obj2;
    return 0;
}
```

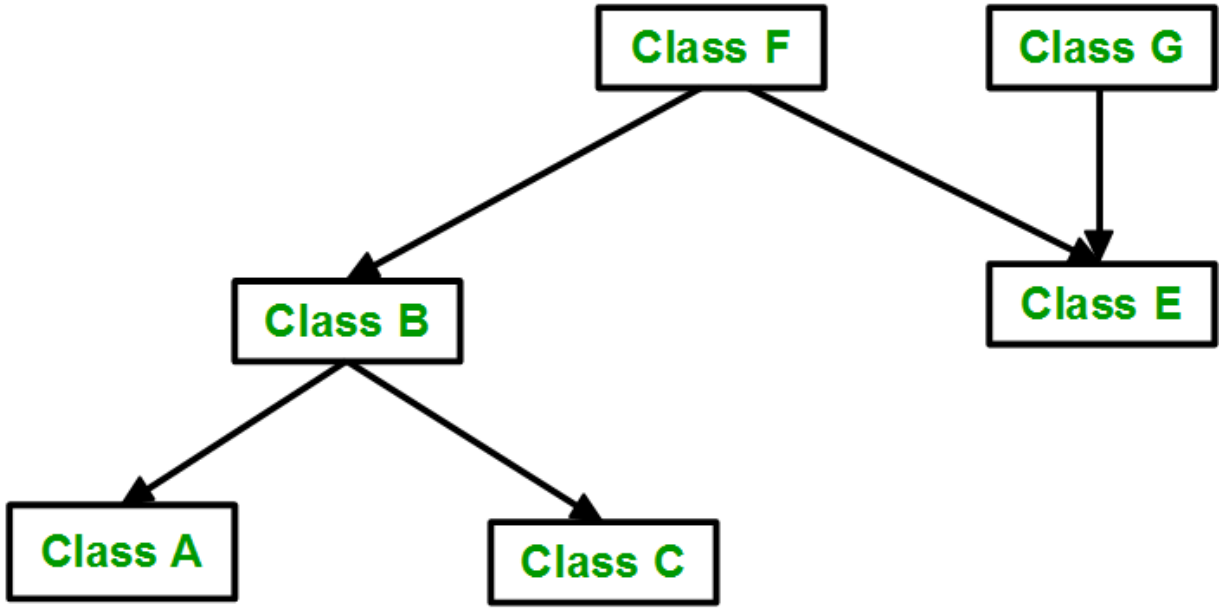
Hierarchical Inheritance: Example

```
This is a Vehicle
This is a bus...
This is a Vehicle
This is a car...
```

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

class Bus: public Vehicle {
  public:
    Bus()
    {
      cout<<"This is a bus..."<<endl;
    }
};

// sub class derived from two base classes
class Car: public Vehicle {
   public:
     Car()
     {
       cout<<"This is a car..."<<endl;
     }
};

// main function
int main() {
    Bus obj1;
    Car obj2;
    return 0;
}
```

Hybrid (Virtual) Inheritance

- Hybrid Inheritance is implemented by combining more than one type of inheritance.
- For example: Combining Hierarchical inheritance and Multiple Inheritance.



Multipath inheritance

- This is a special case of hybrid inheritance.
- A derived class with two base classes and these two base classes have one common base class is called multipath inheritance.
- An ambiguity can arise in this type of inheritance.

Code Reuse

- Inheritance allows us to create a hierarchy of abstract data types that share code.
- The base class defines a general software component from which other specialized classes can be derived.
- Each derived class adds only those members that make it unique.