C++ PROGRAMING

BJARNE STROUSTRUP

C++ Founder

Our Courses

- Master In Flutter Development
- Master In Android Development
- React Native
- Master In Full Stack Development
- Master In Blockchain
- Master In Game Development
- Master In Web Development
- Master In React Native
- Master In Data Science
- Master In ASP.net development
- Laravel
- CodeIgniter
- Node JS
- Angular JS
- React JS

What is C++?

- C++ is an object oriented Mid Level programming language.
- C++ was developed by Bjarne Stroustrup, at Bell Lab, in 1982, as an extension to the C language.
- C++ is a cross-platform language that can be used to create high performance applications.

Difference between C and C++.

- C++ was developed as an extension of C, and both languages have almost the same syntax.
- The main difference between C and C++ is that C++ support classes and objects, while C does not.

What is OOPS?

- OOP stands for Object-Oriented Programming System.
- Object-oriented programming is all about creating objects that contain both data and functions.

Advantages of OOPS.

- OOP is faster and easier to execute
- OOP provides a clear structure for the programs
- OOP helps to keep the C++ code DRY “Don't Repeat Yourself”, and makes the code easier to maintain, modify and debug
- OOP makes it possible to create full reusable applications with
- less code and shorter development time.

OOPS Concept.

1) Class Object
2) Massage Passing
3) Constructor
4) Destructor
5) Polymorphism
6) Inheritance
7) Encapsulation

1) What are Classes and Objects?

Class: - Class is a Collection of Object.

Object: - Object is an Instance of a Class.

Syntax:

```
class ClassName
{
     Access Specifier:
     Data members;
     Member Function()
     {
          //body
     }
};
```

Example:

```
class demo
{
    public:
    int a;
    void myFunction()
    {
        a=10;
    }
};
```

2) Massage Passing.

- Objects communicate with each other by sending and receiving information to each other.

2.1) Pass Value.

- Pass value means by that the compiler copies the value of an argument in a calling function
- A Pass value can be any one of the four variable types: handle, integer, object, or string.

2.2) Return Value.

- A return is a value that a function returns to the calling script or function, when it completes its task.
- A return value can be any one of the four variable types: handle, integer, object, or string.

3) Constructor. (કન્સ્ટ્રક્ટર)

- Constructor in C++ is a special method that is invoked automatically at the time of object creation.
- The purpose of a constructor is to construct an object and assign values to the object's members.
- The constructor in C++ has the same name as the class or structure.
- A constructor has no return type.

There are mainly 3 types of constructors.

3.1) Default
3.2) Parameterized
3.1) Copy constructors

3.1) Default Constructor:

- A constructor with no arguments (or parameters) in the definition is a default constructor.
- Usually, these constructors use to initialize data members (variables) with real values.

Syntax:

```
class ClassName
{
     Access Specifier:
     ClassName()
     {
          //body
     }
};
```

Example:

```
class Demo
{
     public:
     Demo()
     {
        cout<<"Constructor;
     }
};
```

3.2) Parameterized: (પેરામિટરાઇઝ્ડ)

- Unlike the default constructor, it takes parameters (or arguments) in the constructor definition and declaration.
- More than one argument can also be passed to a parameterized constructor.

Syntax:

```
class ClassName
{
     Access Specifier:
     ClassName(parameters)
     {
          //  body
     }
};
```

Example:

```
class Demo
{
    public:
    Demo(int a)
    {
       cout<<"Parameterized;
    }
};
```

3.3) Copy constructors:

- A copy constructor is a member function that initializes an object using another object of the same class.
- It helps to copy data from one object to another.

Syntax:

```
class ClassName
{
     Access Specifier:
     ClassName(ClassName &Object)
     {
          // body
     }
};
```

Example:

```
class Demo
{
     Access Specifier:
     Demo(Demo &Obj)
     {
          cout<<"Copy;
     }
};
```

4) Destructor. (ડિસ્ટ્રક્ટર)

- Destructor is an instance member function which is invoked automatically whenever an object is going to be destroyed. Meaning, a destructor is the last function that is going to be called before an object is destroyed.
- Destructor destroys the class objects created by constructor.
- Destructor has the same name as their class name preceded by a tilde (~) symbol.
- It is not possible to define more than one destructor.
- The destructor is only one way to destroy the object create by constructor. Hence destructor can-not be overloaded.
- Destructor neither requires any argument nor returns any value.

Syntax:

```
class ClassName
{
     Access Specifier:
    ~ClassName()
    {
        // body
    }
}
```

Example:

```
class Demo
{
     public:
    ~Demo()
    {
       cout<<"Destructor;
    }
}
```

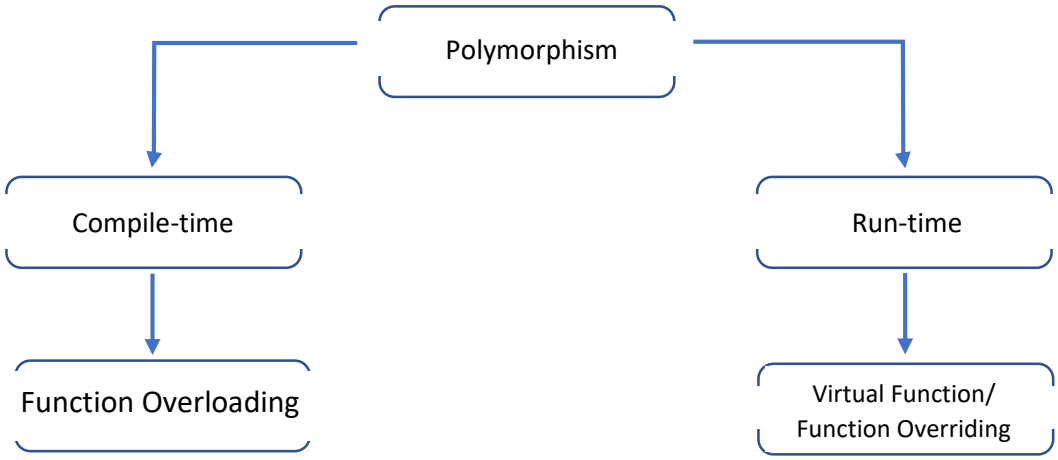
5) Polymorphism. (પોલિમોરફીઝમ)

- Polymorphism mean many form.
- A real-life example of polymorphism is a person who at the same time can have different behavior. A man at the same time is a father, a husband, and an employee. So the same person exhibits different behavior in different situations. This is called polymorphism.

There are two types of polymorphism.

5.1) Compile-time Polymorphism.

5.2) Runtime Polymorphism.



5.1) Compile-time Polymorphism.

- This type of polymorphism is achieved by function overloading or operator overloading.

Function Overloading. (ઓવરલોડિંગ)

- When there are multiple functions with the same name but different parameters, then the functions are said to be overloaded, hence this is known as Function Overloading. Functions can be overloaded by changing the number of arguments or/and changing the type of arguments.

Syntax:

```
class ClassName
{
     Access Specifier:
     Member Function1(Argument1){}

     Member Function1(Argument2){}
};
```

Example:

```
class Demo
{
     public:
     void get(int i){}

     void get(char c){}
};
```

5.2) Run-time Polymorphism.

Function Overriding. (ઓવરરાઇડિંગ)

- Function overriding in C++ is a feature that allows us to use a function in the child class that is already present in its parent class. The child class inherits all the data members, and the member functions present in the parent class.
- If you wish to override any functionality in the child class, then you can implement function overriding. Function overriding means creating a newer version of the parent class function in the child class.

Syntax:

```
class ClassName1
{
     Member Function1() {}
};
class ClassName2: public ClassName1
{
     Member Function1() {}
};
```

Example:

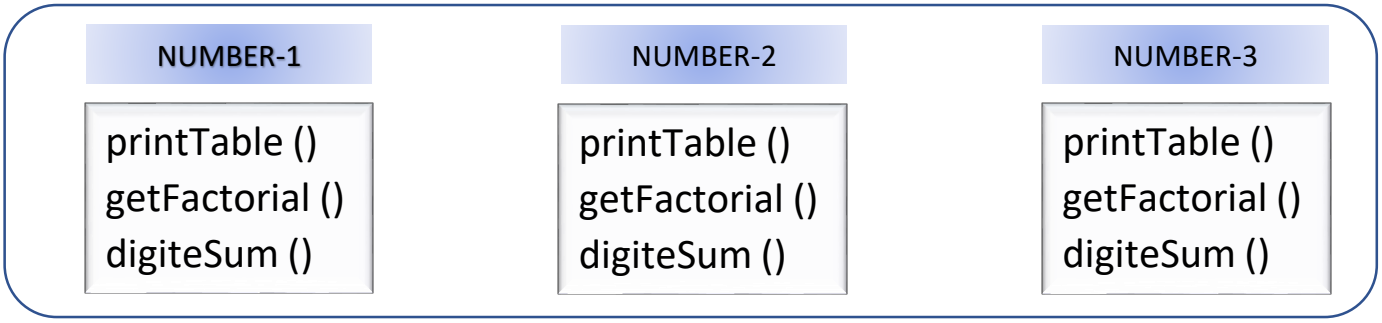
```
class Demo
{
    void get() {}
};
class demo1: public demo
{
    void get() {}
};
```

6) Inheritance. (ઇન્હેરિટન્સ)

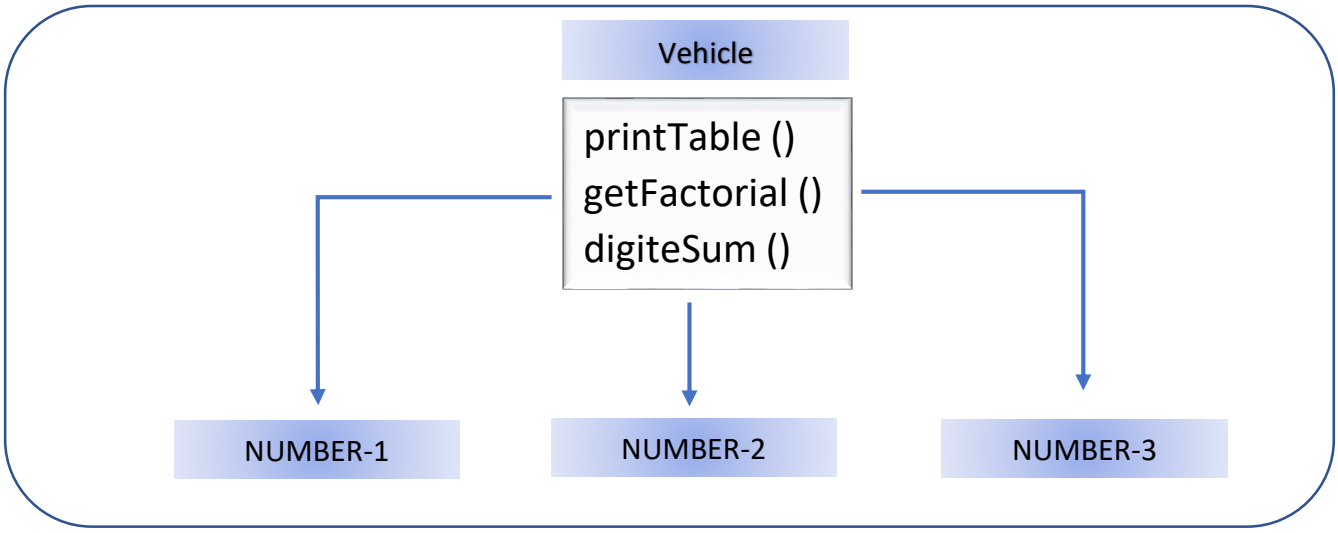
- The ability of a class to inherit properties from another class is called inheritance.
- Inheritance is a process in which, new classes are created from existing classes.
- The new class created is called "sub class", "derived class" or "child class" and the existing class is called "super class", "base class" or "parent class".

Why and when to use inheritance?

- Consider a group of numbers. You need to create classes for NUMBER-1, NUMBER-2 and NUMBER-3. Methods printTable(), getFactorial(), digiteSum() will be same for all three classes. If we create these classes by ignoring inheritance, we have to write all these functions in each of the three classes as shown in the figure below:



- 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 the rest of the three classes from the base class (Vehicle).

Syntax:

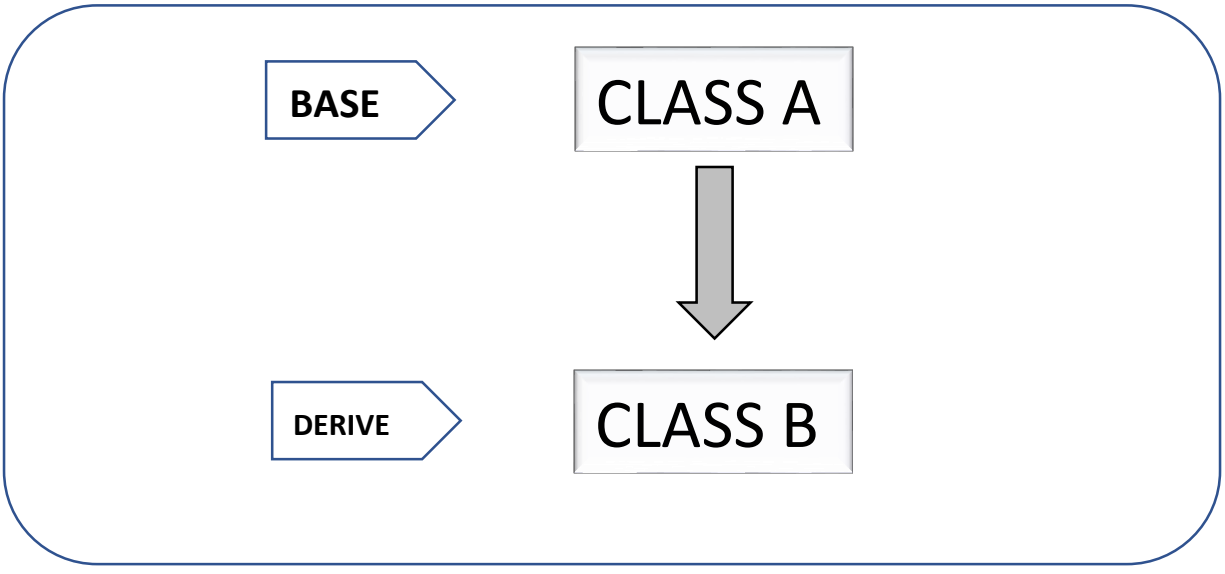
```
class Derived_Class: public Base_Class
{
     //  body;
}
```

Types of Inheritance

6.1) Single Inheritance.
6.2) Multilevel Inheritance.
6.3) Hierarchical inheritance.
6.4) Multiple Inheritance.
6.5) Hybrid Inheritance.

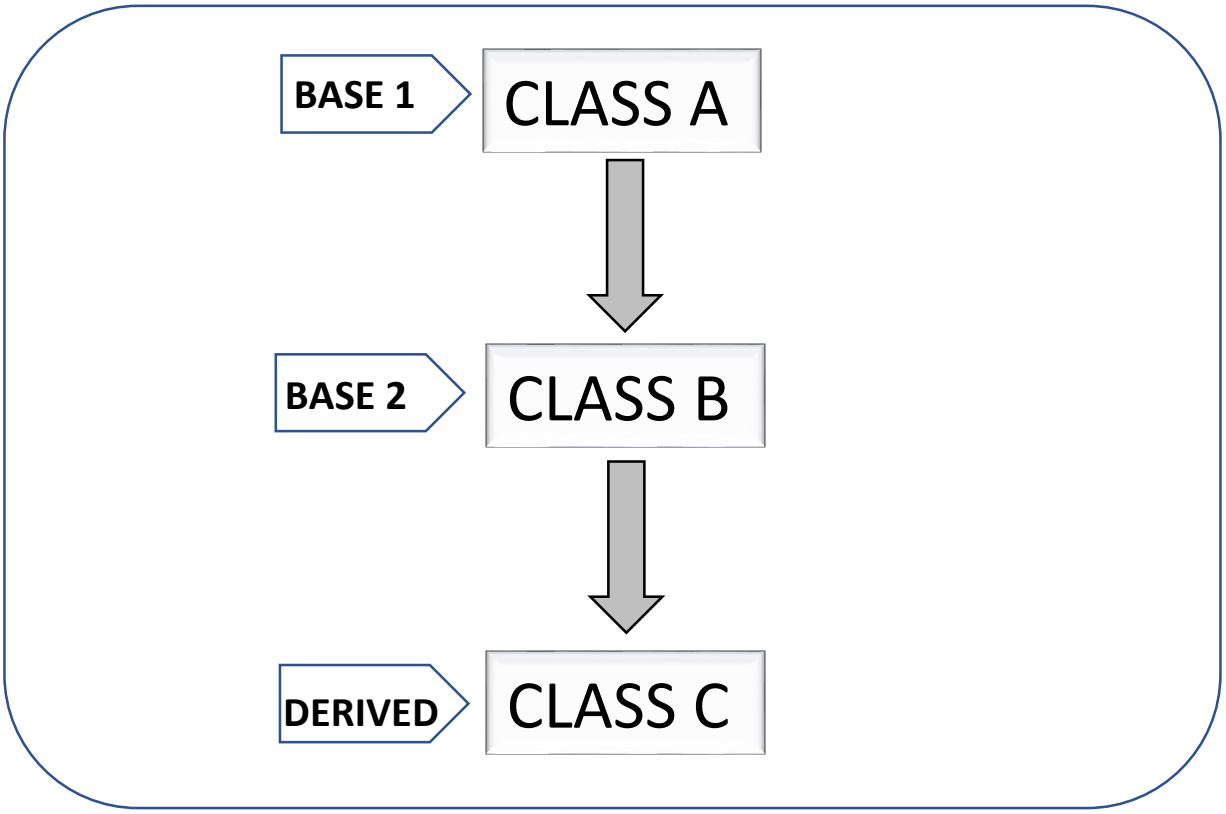
6.1) Single Inheritance.

- In single inheritance, a class is allowed to inherit from only one class. Ex. Subclasses are only inherited by the base class.



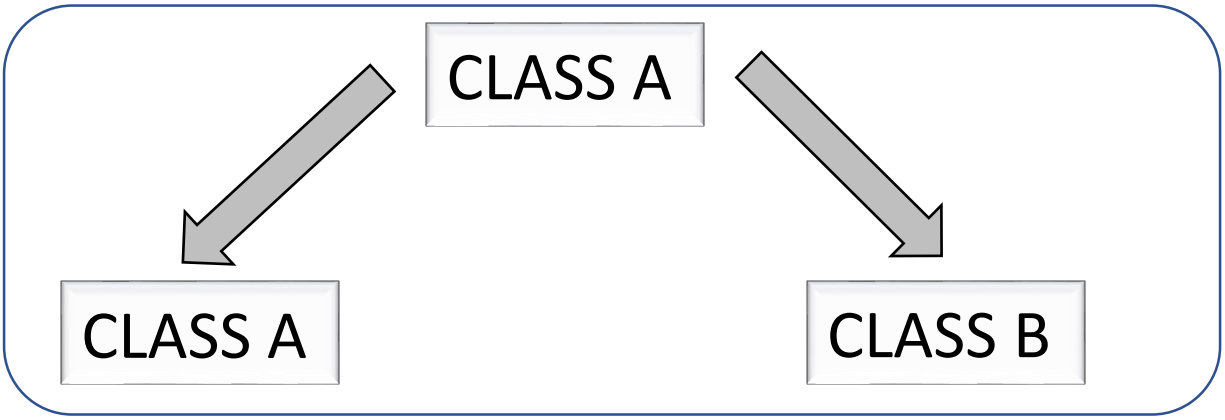
6.2)Multilevel Inheritance.

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



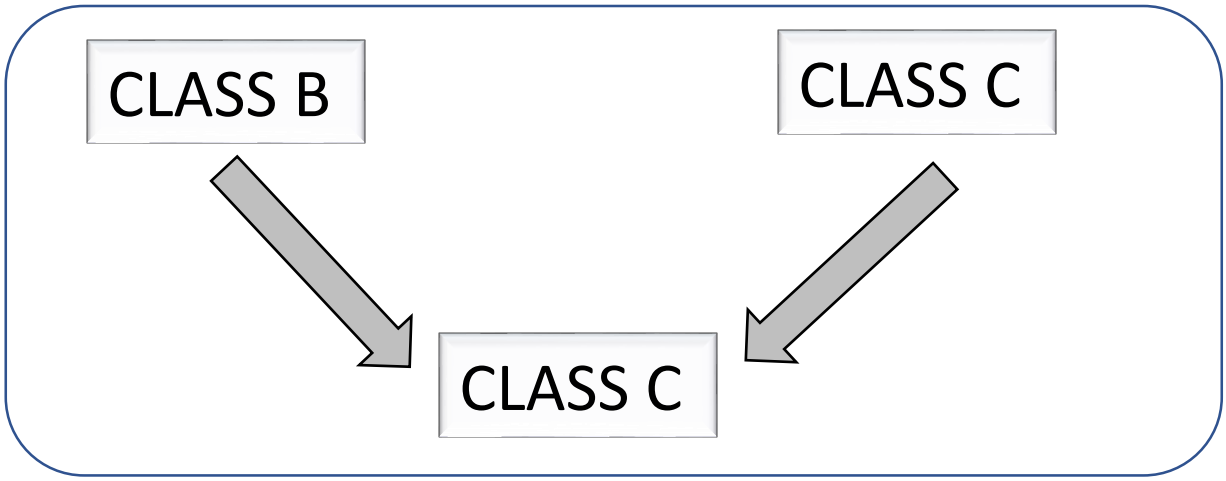
6.3) Hierarchical Inheritance. (હેરારથીકલ)

In this type of inheritance, more than one subclass is inherited from a single base class. Ex. more than one derived class is created from a single base class.



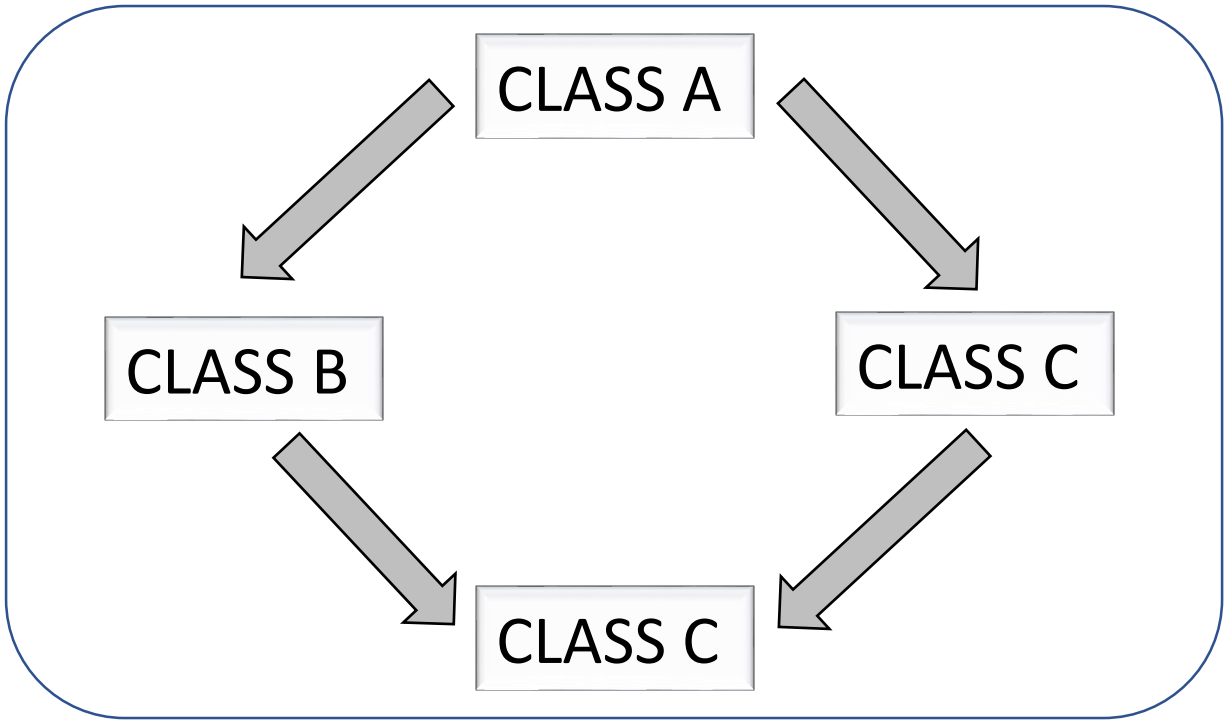
6.4)Multiple Inheritance.

Multiple inheritance can be inherited from more than one class. Ex. a subclass inherits from more than one base class.



6.5) Hybrid Inheritance. (હાયબ્રીડ)

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



Single inheritance

```
class First {
    public:
    int a, b;
    void myFun1() {
        a = 10;
        b = 20;
    }
};
class Second : public First {
    public:
    int sum;
    void myFun2() {
        sum = a + b;
        cout<<"Sum = "<<sum;
    }
};
int main() {
    Second s;
    s.myFun1();
    s.myFun2();
    return 0;
}
```

Multilevel Inheritance

```
class First {
    public:
    int a, b;
    void myFun1() {
        a = 10;
        b = 20;
    }
};
class Second: public First {
    public:
    int sum;
    void myFun2() {
        sum = a + b;
    }
};
class Third: public Second {
    public:
    int per;
    void myFun3() {
        per = sum / 2;
        cout<<"Per = "<<per;
    }
};
int main() {
    Third t;
    t.myFun1();
    t.myFun2();
    return 0;
}
```

Hierarchical inheritance

```
class First {
    public:
    int a, b;
    void myFun1() {
        a = 10;
        b = 20;
    }
};
class Second: public First {
    public:
    int sum;
    void myFun2() {
        sum = a + b;
        cout<<"Sum = "<<sum;
    }
};
class Third: public First {
    public:
    int multi;
    void myFun3() {
        multi = a * b;
        cout<<"Ans = "<<multi;
    }
};
int main() {
    Second s; Third t;
    s.myFun1(); s.myFun2();
    t.myFun1(); t.myFun3();
    return 0;
}
```

Multiple Inheritance

```
class First {
    public:
    int a;
    void myFun1() {
        a = 10;
    }
};
class Second {
    public:
    int b;
    void myFun2() {
        b=20;
    }
};
class Third: public First,
public Second {
    public:
    int sum;
    void myFun3() {
        sum = a + b;
        cout<<"Sum = "<<sum;
    }
};
int main() {
    Third t;
    t.myFun1();
    t.myFun2();
    t.myFun3();
    return 0;
}
```

Hybrid inheritance:

```
class First {
    public:
    int a, b;
    void myFun1() {
        a = 10;
        b = 20;
    }
};
class Second: virtual public First {
    public:
    int sum;
    void myFun2() {
        sum = a + b;
        cout<<"Sum = "<<sum;
    }
};
class Third: virtual public First {
    public:
    int multi;
    void myFun3() {
        multi = a * b;
        cout<<"Multi = "<<multi;
    }
};
```

```
class Fourth: public Second, public Third {
    public:
    int total;
    void myFun4() {
        total = sum + multi;
        cout<<"Total = "<<total;
    }
};
int main() {
    Fourth f;
    f.myFun1();
    f.myFun2();
    f.myFun3();
    f.myFun4();
    return 0;
}
```

7) Encapsulation (એનકેપ્સુલેશન)

- Encapsulation is a method of bundling data and functions
- We cannot directly access any private member from the class. To access the member we need to create a function, called an encapsulation method

```
class ClassName
{
  private:
  datatype variable_name;
  public:
  returntype set(argument-list)
  {
    variable_name = argument;
  }
  return type get()
  {
    return variable_name;
  }
}
int main()
{
  datatype variable_name1;
  ClassName Obj;
  Obj.set(argument);
  variable_name1=Obj.get();
  return 0;
}
```

Example:

```
class Demo {
  private:
  int a;
  public:
  void set(int i) {
    a = i;
  }
  int get() {
    return a;
  }
};
int main() {
  int i;
  Demo d;
  d.set(20);
  i=d.get();
  cout<<"Private int 'a' as 'i' : "<<i;
  return 0;
}
```

8) Friend Class & Function

- A friend class / function can access private and protected members of other classes to which it is declared as a friend.

A) Friend Class :

Example:

```
class First {
  private:
  int a;
  public:
  void myFun1() {
    a = 50;
  }
  friend class Second;
};
class Second {
    public:
    void myFun2(First &f) {
        cout<<"A = "<<f.a;
    }
};
int main() {
  First f;
  f.myFun1();
  Second s;
  s.myFun2(f);
  return 0;
}
```

B) Friend Function :

Example:

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

class First {
  private:
  int a;
  public:
  void myFun1() {
    a = 50;
  }
  friend void myFun2(First);
};
void myFun2(First f) {
    cout<<"A = "<<f.a;
}
int main() {
  First f;
  f.myFun1();
  myFun2(f);
  return 0;
}
```

9) this & static keyword & Scope resolution operator.

A) this keyword:

- This keyword is used to denote an instance of its class

Example:

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

class First {
  private:
  int a;
  public:
  void myFun1(int a) {
    this->a = 50;
    cout<<"A = "<<a;
  }
  void myFun2() {
    cout<<"A = "<<a;
  }
};
int main() {
  First f;
  f.myFun1(20);
  f.myFun2();
  return 0;
}
```

B) static keyword:

- The static keyword has different meanings when used with different types.

I) Static Variables :

Example:

```
class First {
  public:
  void myFun1() {
    static int a=1;
    cout<<" "<<a;
    a++;
  }
};
int main() {
  First f;
  for(int i=1; i<=10; i++) {
    f.myFun1();
  }
  return 0;
}
```

II) Static Members of class :

Example:

```
class Demo
{
    public:
    static void print()
    {
        cout<<"Welcome to
Creative!";
    }
};
int main()
{
    Demo::print();
    return 0;
}
```

C) Scope resolution operator.

- Scope resolution operator is used to denote an instance of its other class

Examples:

```
int a=10;
int main()
{
    int a=20;
    cout<<"Local a :"<<a;
    cout<<"Globle a :"<<::a;
    return 0;
}
```

```
class First
{
    public:
    int a;
    void myFun1() {
        a=20;
    }
};
class Second: public First {
    public:
    int a;
    void myFun2() {
        a=10;
        cout<<"A in Second = "<<a;
        cout<<"A in First = "<<First::a;
    }
};
int main()
{
    Second s;
    s.myFun1();
    s.myFun2();
    return 0;
}
```

```
class First {
    public:
    void myFun();
};
void First::fun() {
    cout << "myFun() called";
}
int main() {
    First f;
    f.fun();
    return 0;
}
```