**Object Oriented Programming in C++**

Object-oriented programming – As the name suggests uses [objects](https://www.geeksforgeeks.org/object-oriented-programming-in-cpp/#objects) in programming. Object-oriented programming aims to implement real-world entities like inheritance, hiding, polymorphism, etc in programming. The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.

**Characteristics of an Object Oriented Programming language**

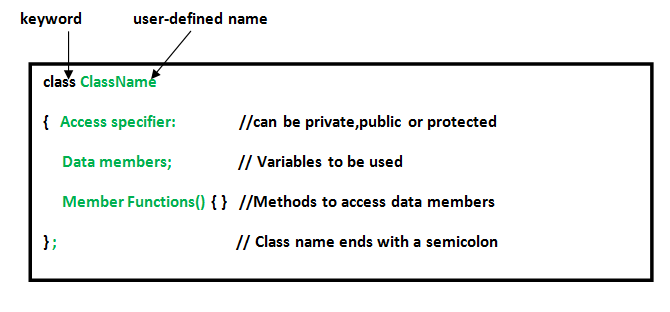
# C++ Classes and Objects

**Class:** A class in C++ is the building block that leads to Object-Oriented programming. It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A C++ class is like a blueprint for an object.  
For Example: Consider the Class of **Cars**. There may be many cars with different names and brand but all of them will share some common properties like all of them will have *4 wheels*, *Speed Limit*, *Mileage range* etc. So here, Car is the class and wheels, speed limits, mileage are their properties.

* A Class is a user defined data-type which has data members and member functions.
* Data members are the data variables and member functions are the functions used to manipulate these variables and together these data members and member functions defines the properties and behavior of the objects in a Class.
* In the above example of class *Car*, the data member will be *speed limit*, *mileage* etc and member functions can be *apply brakes*, *increase speed* etc.

An **Object** is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

**Defining Class and Declaring Objects**

A class is defined in C++ using keyword class followed by the name of class. The body of class is defined inside the curly brackets and terminated by a semicolon at the end.

**Declaring Objects:** When a class is defined, only the specification for the object is defined; no memory or storage is allocated. To use the data and access functions defined in the class, you need to create objects.

**Syntax:**

**ClassName ObjectName;**

**Accessing data members and member functions**: The data members and member functions of class can be accessed using the dot(‘.’) operator with the object. For example if the name of object is *obj* and you want to access the member function with the name *printName()* then you will have to write *obj.printName()* .

**Accessing Data Members**

The public data members are also accessed in the same way given however the private data members are not allowed to be accessed directly by the object. Accessing a data member depends solely on the access control of that data member.  
This access control is given by [Access modifiers in C++](https://www.geeksforgeeks.org/access-modifiers-in-c/). There are three access modifiers : **public, private and protected**.

|  |
| --- |
| // C++ program to demonstrate  // accessing of data members    #include <bits/stdc++.h>  using namespace std;  class Geeks  {      // Access specifier      public:        // Data Members      string geekname;        // Member Functions()      void printname()      {         cout << "Geekname is: " << geekname;      }  };    int main() {        // Declare an object of class geeks      Geeks obj1;        // accessing data member      obj1.geekname = "Abhi";        // accessing member function      obj1.printname();      return 0;  } |

**Output:**

Geekname is: Abhi

**Member Functions in Classes**

There are 2 ways to define a member function:

* Inside class definition
* Outside class definition

To define a member function outside the class definition we have to use the scope resolution :: operator along with class name and function name.

|  |
| --- |
| // C++ program to demonstrate function  // declaration outside class    #include <bits/stdc++.h>  using namespace std;  class Geeks  {      public:      string geekname;      int id;        // printname is not defined inside class definition      void printname();        // printid is defined inside class definition      void printid()      {          cout << "Geek id is: " << id;      }  };    // Definition of printname using scope resolution operator ::  void Geeks::printname()  {      cout << "Geekname is: " << geekname;  }  int main() {        Geeks obj1;      obj1.geekname = "xyz";      obj1.id=15;        // call printname()      obj1.printname();      cout << endl;        // call printid()      obj1.printid();      return 0;  } |

**Output:**

Geekname is: xyz

Geek id is: 15

Note that all the member functions defined inside the class definition are by default **inline**, but you can also make any non-class function inline by using keyword inline with them. Inline functions are actual functions, which are copied everywhere during compilation, like pre-processor macro, so the overhead of function calling is reduced.

Note: Declaring a [friend function](https://www.geeksforgeeks.org/friend-class-function-cpp/) is a way to give private access to a non-member function.

[**Constructors**](https://www.geeksforgeeks.org/constructors-c/)

Constructors are special class members which are called by the compiler every time an object of that class is instantiated. Constructors have the same name as the class and may be defined inside or outside the class definition.  
There are 3 types of constructors:

* [Default constructors](https://www.geeksforgeeks.org/constructors-c/)
* Parameterized constructors
* [Copy constructors](https://www.geeksforgeeks.org/copy-constructor-in-cpp/)

|  |
| --- |
| // C++ program to demonstrate constructors    #include <bits/stdc++.h>  using namespace std;  class Geeks  {      public:      int id;        //Default Constructor      Geeks()      {          cout << "Default Constructor called" << endl;          id=-1;      }        //Parameterized Constructor      Geeks(int x)      {          cout << "Parameterized Constructor called" << endl;          id=x;      }  };  int main() {        // obj1 will call Default Constructor      Geeks obj1;      cout << "Geek id is: " <<obj1.id << endl;        // obj1 will call Parameterized Constructor      Geeks obj2(21);      cout << "Geek id is: " <<obj2.id << endl;      return 0;  } |

**Output:**

Default Constructor called

Geek id is: -1

Parameterized Constructor called

Geek id is: 21

A **Copy Constructor** creates a new object, which is exact copy of the existing object. The compiler provides a default Copy Constructor to all the classes.  
Syntax:

class-name (class-name &){}

[**Destructors**](https://www.geeksforgeeks.org/destructors-c/)

Destructor is another special member function that is called by the compiler when the scope of the object ends.

|  |
| --- |
| // C++ program to explain destructors    #include <bits/stdc++.h>  using namespace std;  class Geeks  {      public:      int id;        //Definition for Destructor      ~Geeks()      {          cout << "Destructor called for id: " << id <<endl;      }  };    int main()    {      Geeks obj1;      obj1.id=7;      int i = 0;      while ( i < 5 )      {          Geeks obj2;          obj2.id=i;          i++;      } // Scope for obj2 ends here        return 0;    } // Scope for obj1 ends here |

**Output:**

Destructor called for id: 0

Destructor called for id: 1

Destructor called for id: 2

Destructor called for id: 3

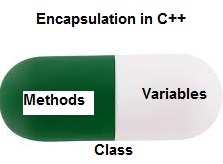
Destructor called for id: 4

Destructor called for id: 7

[Pure Virtual Destructor](https://www.geeksforgeeks.org/pure-virtual-destructor-c/)

# Encapsulation in C++

In normal terms **Encapsulation** is defined as wrapping up of data and information under a single unit. In Object Oriented Programming, Encapsulation is defined as binding together the data and the functions that manipulates them.  
Consider a real life example of encapsulation, in a company there are different sections like the accounts section, finance section, sales section etc. The finance section handles all the financial transactions and keep records of all the data related to finance. Similarly the sales section handles all the sales related activities and keep records of all the sales. Now there may arise a situation when for some reason an official from finance section needs all the data about sales in a particular month. In this case, he is not allowed to directly access the data of sales section. He will first have to contact some other officer in the sales section and then request him to give the particular data. This is what encapsulation is. Here the data of sales section and the employees that can manipulate them are wrapped under a single name “sales section”.



Encapsulation also lead to data abstraction or hiding. As using encapsulation also hides the data. In the above example the data of any of the section like sales, finance or accounts is hidden from any other section.

In C++ encapsulation can be implemented using Class and [access modifiers](https://www.geeksforgeeks.org/access-modifiers-in-c/). Look at the below program:

|  |
| --- |
| // c++ program to explain  // Encapsulation    #include<iostream>  using namespace std;    class Encapsulation  {      private:          // data hidden from outside world          int x;        public:          // function to set value of          // variable x          void set(int a)          {              x =a;          }            // function to return value of          // variable x          int get()          {              return x;          }  };    // main function  int main()  {      Encapsulation obj;        obj.set(5);        cout<<obj.get();      return 0;  } |

output:

5

In the above program the variable **x** is made private. This variable can be accessed and manipulated only using the functions get() and set() which are present inside the class. Thus we can say that here, the variable x and the functions get() and set() are binded together which is nothing but encapsulation.

**Role of access specifiers in encapsulation**

As we have seen in above example, access specifiers plays an important role in implementing encapsulation in C++. The process of implementing encapsulation can be sub-divided into two steps:

1. The data members should be labeled as private using the **private** access specifiers
2. The member function which manipulates the data members should be labeled as public using the **public** access specifier

# Abstraction in C++

Data abstraction is one of the most essential and important feature of object oriented programming in C++. Abstraction means displaying only essential information and hiding the details. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.

Consider a real life example of a man driving a car. The man only knows that pressing the accelerators will increase the speed of car or applying brakes will stop the car but he does not know about how on pressing accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of accelerator, brakes etc in the car. This is what abstraction is.  
 **Abstraction using Classes:** We can implement Abstraction in C++ using classes. Class helps us to group data members and member functions using available access specifiers. A Class can decide which data member will be visible to outside world and which is not.

**Abstraction in Header files:** One more type of abstraction in C++ can be header files. For example, consider the pow() method present in math.h header file. Whenever we need to calculate power of a number, we simply call the function pow() present in the math.h header file and pass the numbers as arguments without knowing the underlying algorithm according to which the function is actually calculating power of numbers.

**Abstraction using access specifiers**

Access specifiers are the main pillar of implementing abstraction in C++. We can use access specifiers to enforce restrictions on class members. For example:

* Members declared as **public** in a class, can be accessed from anywhere in the program.
* Members declared as **private** in a class, can be accessed only from within the class. They are not allowed to be accessed from any part of code outside the class.

We can easily implement abstraction using the above two features provided by access specifiers. Say, the members that defines the internal implementation can be marked as private in a class. And the important information needed to be given to the outside world can be marked as public. And these public members can access the private members as they are inside the class.

**Example**:

|  |
| --- |
| #include <iostream>  using namespace std;    class implementAbstraction  {      private:          int a, b;        public:            // method to set values of          // private members          void set(int x, int y)          {              a = x;              b = y;          }            void display()          {              cout<<"a = " <<a << endl;              cout<<"b = " << b << endl;          }  };    int main()  {      implementAbstraction obj;      obj.set(10, 20);      obj.display();      return 0;  } |

Output:

a = 10

b = 20

You can see in the above program we are not allowed to access the variables a and b directly, however one can call the function set() to set the values in a and b and the function display() to display the values of a and b.

**Advantages of Data Abstraction**:

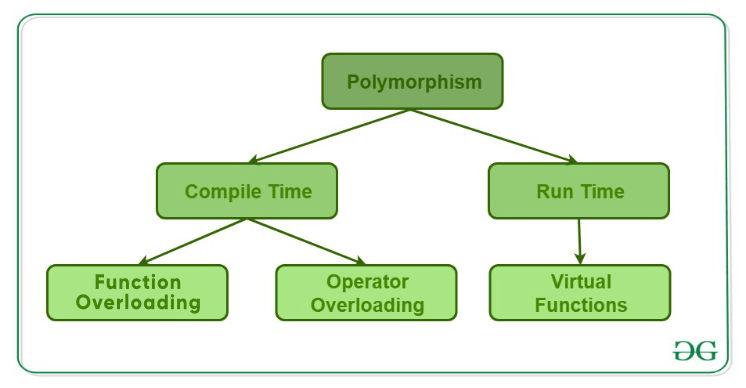
* Helps the user to avoid writing the low level code
* Avoids code duplication and increases reusability.
* Can change internal implementation of class independently without affecting the user.
* Helps to increase security of an application or program as only important details are provided to the user.

# Polymorphism in C++

The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form. A real-life example of polymorphism, a person at the same time can have different characteristics. Like a man at the same time is a father, a husband, an employee. So the same person posses different behavior in different situations. This is called polymorphism. Polymorphism is considered as one of the important features of Object Oriented Programming.

**In C++ polymorphism is mainly divided into two types:**

* Compile time Polymorphism
* Runtime Polymorphism



Types of Polymorphism

1. **Compile time polymorphism**: This type of polymorphism is achieved by function overloading or operator overloading.
   * [**Function Overloading**](https://www.geeksforgeeks.org/function-overloading-c/): When there are multiple functions with same name but different parameters then these functions are said to be **overloaded**. Functions can be overloaded by **change in number of arguments** or/and **change in type of arguments**.  
     [Rules of Function Overloading](https://www.geeksforgeeks.org/function-overloading-in-c/)

|  |
| --- |
| // C++ program for function overloading  #include <bits/stdc++.h>    using namespace std;  class Geeks  {      public:        // function with 1 int parameter      void func(int x)      {          cout << "value of x is " << x << endl;      }        // function with same name but 1 double parameter      void func(double x)      {          cout << "value of x is " << x << endl;      }        // function with same name and 2 int parameters      void func(int x, int y)      {          cout << "value of x and y is " << x << ", " << y << endl;      }  };    int main() {        Geeks obj1;        // Which function is called will depend on the parameters passed      // The first 'func' is called      obj1.func(7);        // The second 'func' is called      obj1.func(9.132);        // The third 'func' is called      obj1.func(85,64);      return 0;  } |

* + **Output:**
  + value of x is 7
  + value of x is 9.132
  + value of x and y is 85, 64
  + In the above example, a single function named *func* acts differently in three different situations which is the property of polymorphism.
  + [**Operator Overloading**](https://www.geeksforgeeks.org/operator-overloading-c/): C++ also provide option to overload operators. For example, we can make the operator (‘+’) for string class to concatenate two strings. We know that this is the addition operator whose task is to add two operands. So a single operator ‘+’ when placed between integer operands , adds them and when placed between string operands, concatenates them.  
    **Example**:

|  |
| --- |
| // CPP program to illustrate  // Operator Overloading  #include<iostream>  using namespace std;    class Complex {  private:      int real, imag;  public:      Complex(int r = 0, int i =0)  {real = r;   imag = i;}        // This is automatically called when '+' is used with      // between two Complex objects      Complex operator + (Complex const &obj) {           Complex res;           res.real = real + obj.real;           res.imag = imag + obj.imag;           return res;      }      void print() { cout << real << " + i" << imag << endl; }  };    int main()  {      Complex c1(10, 5), c2(2, 4);      Complex c3 = c1 + c2; // An example call to "operator+"      c3.print();  } |

* + Output:
  + 12 + i9
  + In the above example the operator ‘+’ is overloaded. The operator ‘+’ is an addition operator and can add two numbers(integers or floating point) but here the operator is made to perform addition of two imaginary or complex numbers. To learn operator overloading in details visit [this](https://www.geeksforgeeks.org/operator-overloading-c/) link.

1. [**Runtime polymorphism**](https://www.geeksforgeeks.org/virtual-functions-and-runtime-polymorphism-in-c-set-1-introduction/): This type of polymorphism is achieved by Function Overriding.
   * [**Function overriding**](https://www.geeksforgeeks.org/override-keyword-c/) on the other hand occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be **overridden**.

|  |
| --- |
| // C++ program for function overriding    #include <bits/stdc++.h>  using namespace std;    class base  {  public:      virtual void print ()      { cout<< "print base class" <<endl; }        void show ()      { cout<< "show base class" <<endl; }  };    class derived:public base  {  public:      void print () //print () is already virtual function in derived class, we could also declared as virtual void print () explicitly      { cout<< "print derived class" <<endl; }        void show ()      { cout<< "show derived class" <<endl; }  };    //main function  int main()  {      base \*bptr;      derived d;      bptr = &d;        //virtual function, binded at runtime (Runtime polymorphism)      bptr->print();        // Non-virtual function, binded at compile time      bptr->show();        return 0;  } |

* + Output:
  + print derived class
  + show base class
  + To learn runtime polymorphism in details visit [this](https://www.geeksforgeeks.org/virtual-functions-and-runtime-polymorphism-in-c-set-1-introduction/) link.

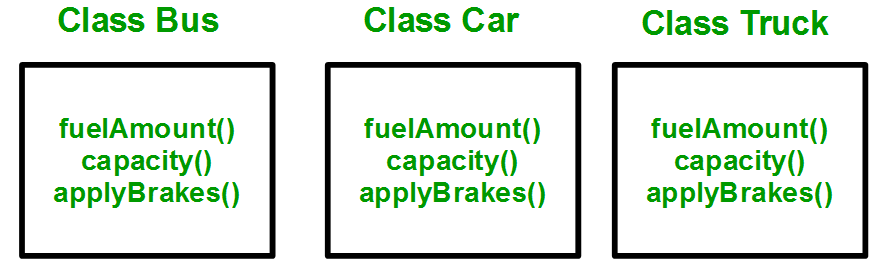
# Inheritance in C++

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.   
**The article is divided into following subtopics:**

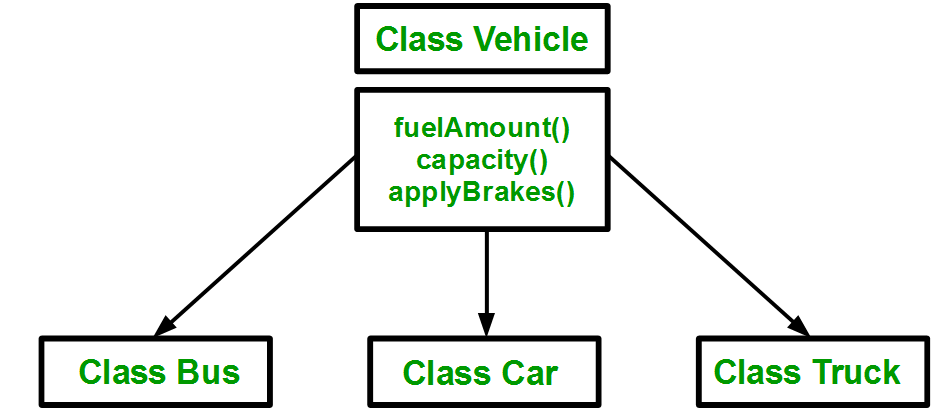
1. [Why and when to use inheritance?](https://www.geeksforgeeks.org/inheritance-in-c/#Why and when to use inheritance?)
2. [Modes of Inheritance](https://www.geeksforgeeks.org/inheritance-in-c/#Modes of Inheritance)
3. [Types of Inheritance](https://www.geeksforgeeks.org/inheritance-in-c/#Types of Inheritance)

**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.   
**Note**: A derived class doesn’t inherit ***access*** to private data members. However, it does inherit a full parent object, which contains any private members which that class declares.

|  |
| --- |
| // C++ program to demonstrate implementation  // of Inheritance    #include <bits/stdc++.h>  using namespace std;    // 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 << '\n';      cout << "Parent id is: " <<  obj1.id\_p << '\n';        return 0;  } |

**Output**

Child id is 7

Parent id is 91

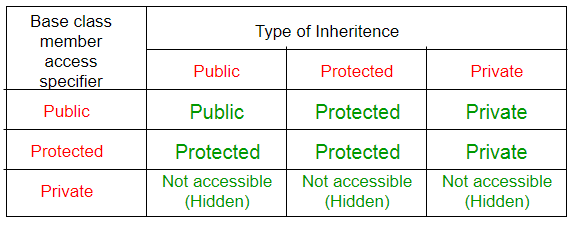
In the above program the ‘Child’ class is publicly inherited from the ‘Parent’ class so the public data members of the class ‘Parent’ will also be inherited by the class ‘Child’.  
 **Modes of Inheritance**

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.

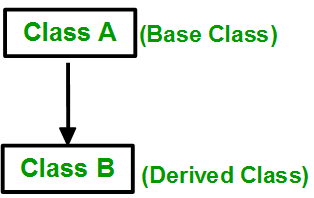
|  |
| --- |
| // C++ Implementation to show that a derived class  // doesn’t inherit access to private data members.  // However, it does inherit a full parent object.  class A  {  public:      int x;  protected:      int y;  private:      int z;  };    class B : public A  {      // x is public      // 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  }; |

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:



**Types of Inheritance in C++**

**1. 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

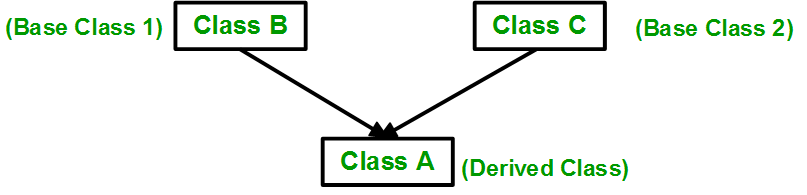
};

|  |
| --- |
| // C++ program to explain  // Single inheritance  #include<iostream>  using namespace std;    // base class  class Vehicle {    public:      Vehicle()      {        cout << "This is a Vehicle\n";      }  };    // sub class derived from a single 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

**2. 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**.



**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.

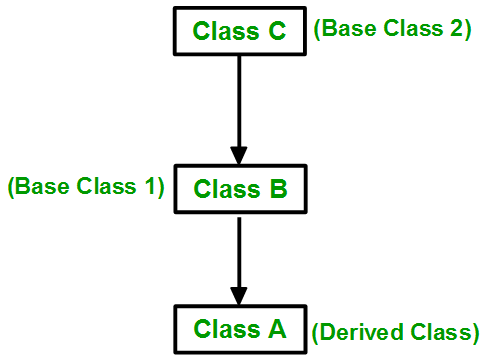
|  |
| --- |
| // C++ program to explain  // multiple inheritance  #include<iostream>  using namespace std;    // first base class  class Vehicle {    public:      Vehicle()      {        cout << "This is a Vehicle\n";      }  };    // second base class  class FourWheeler {    public:      FourWheeler()      {        cout << "This is a 4 wheeler Vehicle\n";      }  };    // 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;  } |

**Output**

This is a Vehicle

This is a 4 wheeler Vehicle

Please visit [this](https://www.geeksforgeeks.org/multiple-inheritance-in-c/) link to learn multiple inheritance in details.   
3. **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\n";      }  };    // first sub\_class derived from class vehicle  class fourWheeler: public Vehicle  {  public:      fourWheeler()      {        cout << "Objects with 4 wheels are vehicles\n";      }  };  // sub class derived from the derived base class fourWheeler  class Car: public fourWheeler {     public:       Car()       {         cout << "Car has 4 Wheels\n";       }  };    // 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.



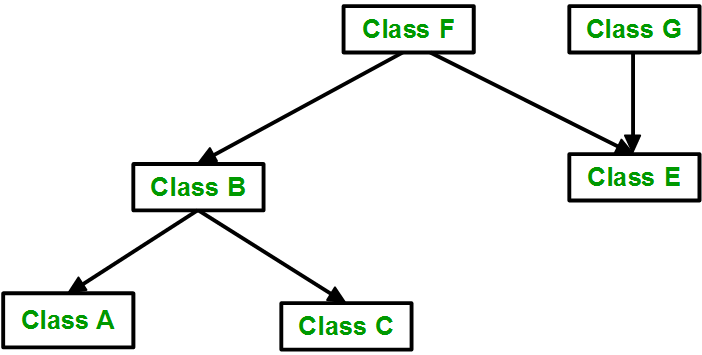
|  |
| --- |
| // C++ program to implement  // Hierarchical Inheritance  #include<iostream>  using namespace std;    // base class  class Vehicle  {    public:      Vehicle()      {        cout << "This is a Vehicle\n";      }  };      // first sub class  class Car: public Vehicle  {    };    // second 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

**5. 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\n";      }  };    //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;  } |

**Output**

This is a Vehicle

Fare of Vehicle

**6. A special case of hybrid inheritance: Multipath 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 arrise in this type of inheritance. 

Consider the following program:

|  |
| --- |
| // C++ program demonstrating ambiguity in Multipath  // Inheritance    #include<iostream>    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;                  // Statement 1, Error      // obj.a = 100;                 // Statement 2, Error        obj.ClassB::a = 10;  // Statement 3      obj.ClassC::a = 100; // Statement 4        obj.b = 20;      obj.c = 30;      obj.d = 40;        cout << " a from ClassB  : " << obj.ClassB::a;      cout << "\n a from ClassC  : " << obj.ClassC::a;        cout << "\n b : " << obj.b;      cout << "\n c : " << obj.c;      cout << "\n d : " << obj.d << '\n';  } |

**Output:**

a from ClassB : 10

a from ClassC : 100

b : 20

c : 30

d : 40

In the above example, both ClassB and ClassC inherit ClassA, they both have single copy of ClassA. However ClassD inherit both ClassB and 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:   
**Avoiding ambiguity using scope resolution operator:**   
Using scope resolution operator we can manually specify the path from which data member a will be accessed, as shown in statement 3 and 4, in the above example.

|  |
| --- |
| obj.ClassB::a = 10;       // Statement 3  obj.ClassC::a = 100;      // Statement 4 |

Note: Still, there are two copies of ClassA in ClassD.  
**Avoiding ambiguity using virtual base class:**

|  |
| --- |
| #include<iostream>    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;       // Statement 3      obj.a = 100;      // Statement 4        obj.b = 20;      obj.c = 30;      obj.d = 40;        cout << "\n a : " << obj.a;      cout << "\n b : " << obj.b;      cout << "\n c : " << obj.c;      cout << "\n d : " << obj.d << '\n';  } |

**Output:**

a : 100

b : 20

c : 30

d : 40

According to the above example, ClassD has only one copy of ClassA, therefore, statement 4 will overwrite the value of a, given at statement 3.