Inheritance

- The capability of a class to derive properties and characteristics from another class is called Inheritance.
  - **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.
- Following are the properties which a derived class doesn't inherit from its parent class:
  - The base class's constructors and destructor.
  - The base class's friend functions.
  - Overloaded operators of the base class.

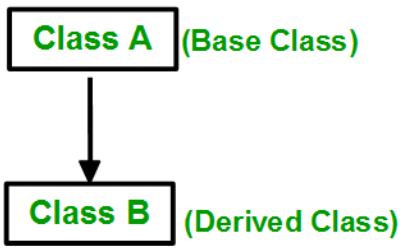
- **Modes of Inheritance:**
  - **Public mode:** If we derive a sub class from a public base class. Then the public member of the base class will become public in the derived class and protected members of the base class will become protected in derived class.
  - **Protected mode:** If we derive a sub 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.
  - Note: The private members in the base class cannot be directly accessed in the derived class, while protected members can be directly accessed.

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)

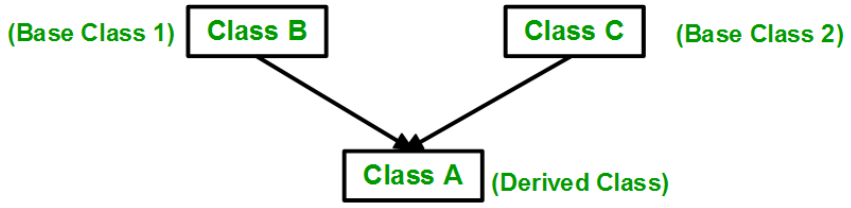
- **Example:**

```
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
};
```

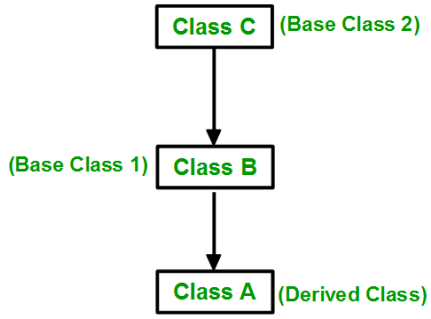
- **Types of Inheritance:**
  - **Single Inheritance**: A class is allowed to inherit from only one class. i.e., one sub class is inherited by one base class only.



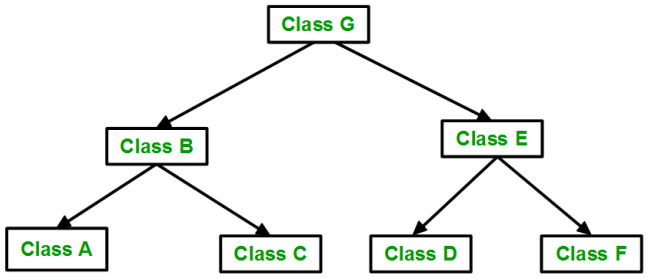
- **Multiple Inheritance:** A class can inherit from more than one classes. i.e., one sub class is inherited from more than one base classes.



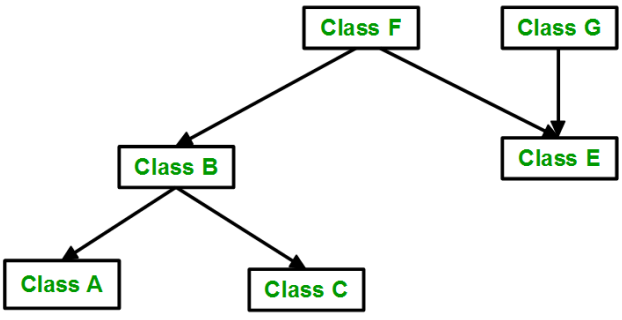
- **Multilevel Inheritance**: A derived class is created from another derived class.



- **Hierarchical 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.



- **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 (*A special case of hybrid inheritance)*:**
  - A derived class with two base classes and these two base classes has one common base class is called multipath inheritance.
  - An ambiguity can arise in this type of inheritance, i.e., called *Diamond Problem*.
  - **Example:**

Input:

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

class ClassA
{
   public:
      ClassA()
      {
          cout << "ClassA constructor" << endl;
      }
      int a;
};

class ClassB : public ClassA
{
   public:
      ClassB()
      {
         cout << "ClassB constructor" << endl;
      }
      int b;
};

class ClassC : public ClassA
{
   public:
      ClassC()
      {
         cout << "ClassC constructor" << endl;
      }
      int c;
};

class ClassD : public ClassB, public ClassC
{
   public:
      ClassD()
      {
         cout << "ClassD constructor" << endl;
      }
```

```
    int d;
};

int main()
{
   ClassD obj;

   // obj.a = 10;        // Error statement
   // obj.a = 100;       // Error statement

   obj.ClassB::a = 10;
   obj.ClassC::a = 100;

   obj.b = 20;
   obj.c = 30;
   obj.d = 40;

   cout << "A from ClassB : " << obj.ClassB::a << endl;
   cout << "A from ClassC : " << obj.ClassC::a << endl;

   cout << "B : " << obj.b << endl;
   cout << "C : " << obj.c << endl;
   cout << "D : " << obj.d << endl;

   return 0;
}
```

Output:

```
ClassA constructor
ClassB constructor
ClassA constructor
ClassC constructor
ClassD constructor
A from ClassB : 10
A from ClassC : 100
B : 20
C : 30
D : 40
```

Explanation: 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, because compiler can't differentiate between two copies of ClassA in ClassD.

- There are 2 ways to avoid this ambiguity:
  1. **Avoiding ambiguity using scope resolution operator:**
     - Using scope resolution operator, we can manually specify the path from which data member a will be accessed.

```
obj.ClassB::a = 10;
obj.ClassC::a = 100;
```

     - Note: Still, there are two copies of ClassA in ClassD.

  2. **Avoiding ambiguity using virtual base class:**

Input:

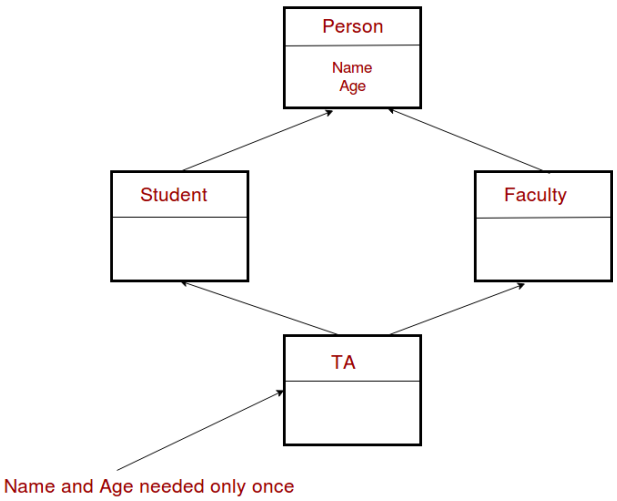
```
#include <iostream>
using namespace std;

class ClassA
{
   public:
      ClassA()
      {
          cout << "ClassA constructor" << endl;
      }
      int a;
};

class ClassB : virtual public ClassA
{
   public:
      ClassB()
      {
         cout << "ClassB constructor" << endl;
      }
      int b;
};

class ClassC : virtual public ClassA
{
   public:
      ClassC()
      {
         cout << "ClassC constructor" << endl;
      }
      int c;
};

class ClassD : public ClassB, public ClassC
```

```
{
   public:
      ClassD()
      {
         cout << "ClassD constructor" << endl;
      }
      int d;
};

int main()
{
   ClassD obj;

   obj.a = 10;
   obj.b = 20;
   obj.c = 30;
   obj.d = 40;

   cout << "A : " << obj.a << endl;
   cout << "B : " << obj.b << endl;
   cout << "C : " << obj.c << endl;
   cout << "D : " << obj.d << endl;

   return 0;
}
```

Output:

```
ClassA constructor
ClassB constructo
ClassC constructor
ClassD constructor
A : 10
B : 20
C : 30
D : 40
```

- **Diamond Problem:**
  - The diamond problem occurs when two super classes 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.



  - In the above diagram, constructor of 'Person' is called two times. Destructor of 'Person' will also be called two times when object of TA is destructed. So, object of TA has two copies of all members of 'Person', this causes ambiguities.
  - The solution to this problem is *'virtual'* keyword. We make the classes 'Faculty' and 'Student' as virtual base classes to avoid two copies of 'Person' in 'TA' class.
  - After that constructor of 'Person' is called once. One important thing to note that when we use 'virtual' keyword, the default constructor of grandparent class is called by default even if the parent classes explicitly call parameterized constructor.
  - 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.

- **Object Slicing:**
  - A derived class object can be assigned to a base class object, but the other way is not possible.
  - Object slicing happens when a derived class object is assigned to a base class object, additional attributes of a derived class object are sliced off to form the base class object.
  - **Example:**

```
class Base { int x, y; };

class Derived : public Base { int z, w; };

int main()
```

```
{
   Derived d;
   Base b = d;      // Object Slicing,  z and w of d are sliced off
}
```

- We can avoid above unexpected behavior with the use of pointers or references. Object slicing doesn't occur when pointers or references to objects are passed as function arguments since a pointer or reference of any type takes same amount of memory.
- Object slicing can be prevented by making the base class function pure virtual there by disallowing object creation. It is not possible to create the object of a class which contains a pure virtual method.

- **Hiding of all overloaded methods with same name in base class:**
  - If a derived class redefines base class member method, then all the base class methods with same name become hidden in derived class.
  - Even if the signature of the derived class method is different, all the overloaded methods in base class become hidden.
  - Note: These facts are true for both static and non-static methods.
  - **Example:**

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

class Base
{
    public:
      int fun()
      {
         cout << "Base::fun() called";
      }
      int fun(int i)
     {
         cout << "Base::fun(int i) called";
     }
};

class Derived : public Base
{
    public:
   // Makes Base::fun() and Base::fun(int ) hidden
   int fun(char c)
   {
      cout << "Derived::fun(char c) called";
   }
```

```
};

int main()
{
    Derived d;
    d.fun();        // Compiler error
    d.fun(1);       // No  compiler error, but call Derived::fun(char c)
    d.fun('a');     // This is fine
    return 0;
}
```

- There is a way mitigate this kind of issue. If we want to overload a function of a base class, it is possible to unhide it by using the *'using'* keyword.
- Declare *Base::fun* in Derived class.

```
using Base::fun;
```

- After this there will be no compilation error and we will get the desired output of above code.

```
Base::fun() called
Base::fun(int i) called
Derived::fun(char c) called
```

- **final class:**
  - This design makes the class which can't be inherited.
  - **Using 'final' keyword:**
    - In C++ 11 we can make the base class non-inheritable by using *final* specifier.

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

class Base final
{
    // body
};
class Derived : public Base     // compile error because base class is final
{
    // body
};

int main()
{
    return 0;
}
```

- **Without using 'final' keyword:**
  - It makes use of private constructor, virtual inheritance and friend class.
  - An extra class *MakeFinal* (whose default constructor is private) is used for our purpose.
  - Constructor of *Final* can call private constructor of *MakeFinal* as *Final* is a friend of *MakeFinal* .
  - Note that *MakeFinal* is also a virtual base class. The reason for this is to call the constructor of *MakeFinal* through the constructor of *Derived*, not *Final* (The constructor of a virtual base class is not called by the class that inherits from it, instead the constructor is called by the constructor of the concrete class).
  - **Example:**

    Input:

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

class Final;      // The class to be made final

class MakeFinal     // used to make the Final class final
{
   private:
      MakeFinal() { cout << "MakFinal constructor" << endl; }
      friend class Final;
};

class Final : virtual MakeFinal
{
   public:
      Final() { cout << "Final constructor" << endl; }
};

class Derived : public Final     // Compiler error
{
   public:
      Derived() { cout << "Derived constructor" << endl; }
};

int main(int argc, char *argv[])
{
   Derived d;
   return 0;
}
```

Output:

```
In constructor 'Derived::Derived()':
error: 'MakeFinal::MakeFinal()' is private
```

Explanation:

In the above example, *Derived*'s constructor directly invokes *MakeFinal's* constructor, and the constructor of *MakeFinal* is private, therefore we get the compilation error.