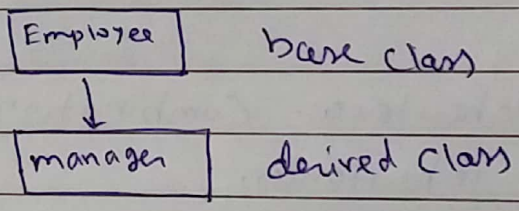
Inheritance is the process of taking some or all the features of an existing entity to build a new entity.

~~In case of C++~~, For example, mobile phones have the features of landline phones such as dialing and receiving a call. But, apart from that, it has some more features such as messaging, maintaing a phonebook etc.

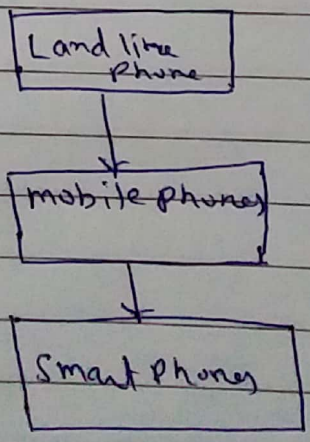
~~In case of C++~~, C++ also provides the facility of inheriting the features of an existing class to a new class. In this case, the new existing class is called as base class and new class is called as derived class. For example, here Landline phones will be base class and mobile phones will be derived class.

Types of inheritance

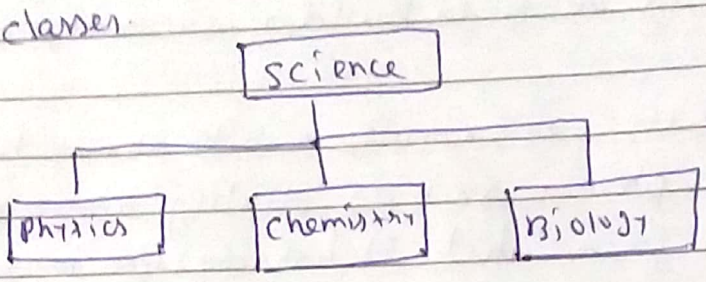
(i) Single inheritance - only one base class and one derived class.



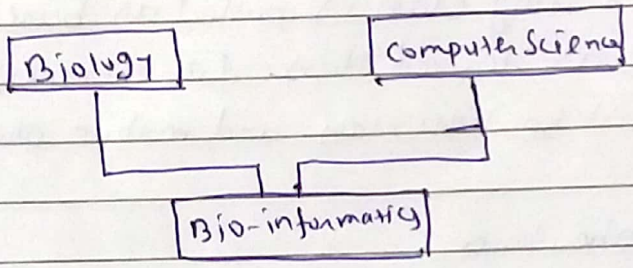
(ii) multi-level inheritance :- when a derived class may act as a base class for another derived class



(iii) hierarchical inheritance:- when a base class has multiple derived classes.



(iv) multiple inheritance:- when a derived class inherits features from multiple base classes.



(v) hybrid inheritance - Combination of any above two mentioned inheritances.

=> how to declare a derived class which is inherited from a base class.

```
class abc
        { int a;
          public:
          void fun1();
        };

class pqr : public abc
        {   int b;
            public:
            void fun2();
        };
```

here, abc is the base class and pqr is derived class. from class abc.

Here, class pqr : public abc

In the above line, if we replace the word 'public' with 'private' or 'protected' then nature of inheritance will change.

-) Derive class inherits all data members of base class
-) Derive class may add data members of its own.
-) Derived class cannot access private members (whether data member or member function) of base class

-) Derived class inherits all member functions of base class.

Here, Public data members of base class will also became public in derived class.

```
class abc
{   int a;
    void Print_a() { cout << a; }
Public:
    void Set_a(int x)
    { a=x;
    }

int ~~void~~ fun1(int x, int y)
    { int z;
      z = x*y;
      return z;
    }
void Print_member(~~int~~)
    {
      Print_a();
    }
class xyz
{
};
int b;
```

~~class xyz : public abc {~~
~~int b;~~

```
class xyz : public abc
{
    int b;

    public:
        void set_b(int y)
        {
            b = y;
        }
        void setmembers (int p, int q)
        {
            a = p;
            b = q;
        }

    int fun2 (int x, int y)
        {
            int z;
            z = x+y;
            return z;
        }
```

```
int main()
{
    abc m;
    m.set_a(10);          // m.a = 10
    m.print_member();     // it will call internally
                          Print_a() which is a
                          Private member func^n.

    int c;
    c = m.fun1(20,30);
    cout << c;            // it will print 600

    xyz n;

    n.set_b(20);          // n.b = 20
    n.Setmember(10,20);
/* it will give error because a=P is not possible
because 'a' is a private data member of ~~class~~ base
class abc. */
    n.Set_a(10);
/* it will not give error because set_a() is a public
member func^n of base class which can be accessed
by the object of derived class. Here ~~Set a() internally~~
~~calls~~ Here, value of 'a' will be set as 10. */

    int d;
    d = n.fun1(50,60);
    cout << d;            // it will print 3000

    d = n.fun2(200,300);
    cout << d             // it will print 500
```

here fun1() is public member func^n of base class abc. Hence, derived class xyz can also access this function. As a result, 30 is being printed.

Overloading and Overriding of a function in derived class

```
void fun1 (int x, int y);
void fun1 (int x, int y);
```

In the above case both the func^{n}s have same return types, same name, and same arguments. In other words, both have same signatures. This is the situation of func^n overriding.

```
void fun1 (int x, int y);
void fun1 (float x, float y);
```

Here both the func^{n}s differ in datatypes of the arguments. This the case of func^n overloading.

Inheritance allows both func^n overloading & overriding. If a func^n fun1() is in base class and also exists in derived class with same signature then if an object of the base class calls invokes this func^n then its body in base class will be executed but if an object of derived class invokes the func^n then its body in the derived class will be executed.

Similarly, in case of func^n overloading also, if a func^n in base class and also exists in derived class, making a situ^n of func^n overloading then its body in the base class will be executed if invoked by an object of base class & its body in the derived class will be executed if invoked by an object of derived class.

```
class abc
  { int a;
    Public:
      void Set_a (int n)
        { a=n;
        }
      void print ( )
        {
          cout << a;
        }
      void add (int n)
        { int y=2;
          cout << n+y;
        }
  };

class xyz
  { int b;
    Public:
      void Set_b (int y)
        { b=y;
        }
      void print ( )
        { cout << b;
        }
      void add ( int n, int y)
        { cout << n+y;
        }
};
```

```
int main()
    {
        abc m;
        m.set_a(10);                    // m.a = 10
        m.Print();
// it will execute the body of Print() which is written within
the 'abc' class bcoz 'm' is an object of 'abc' class.
Hence, 10 will be printed. */

        m.add(10);
// Here also, body of add() in the class 'abc' willbe executed
bcoz add() is being invoked by an object of class 'abc'.
Hence, 12 will be printed. */

        xyz n;
        n.set_b(20);                    // n.b = 20
        n.Print();
// it will execute the body of Print() which is written
within 'xyz' class bcoz 'n' is an object of 'xyz' class.
It is an example of func overriding. */
        n.add(50,60);
// It will execute the body of add() written in 'xyz' class
bcoz 'n' is an object of 'xyz' class. It is the case of
func overloading. Hence, 110 will be printed. */
```

Protected

Apart from public and private, C++ provides a third access specifier k/a 'Protected'.

As we know that, if a member is declared as 'private' then it cannot be accessed by the members of ~~der~~ func^n of the derived class. So, to solve the problem, 'protected' specifier has been provided. If a member (data member or member func^n) is declared as 'protected' then it can be accessed by member func's of ~~both~~ the same class as well as its ~~derived class~~ immediate derived class but it cannot be accessed by an functions outside these two classes.

```
class abc
{   protected:
    int a;
    public:
    void set( int n)
        { a=n;
        }
    void Print ( )
        { cout << a;
        }
};
class xyz : public abc
{   int b;
    public:
        void set (int n, int y)
            { a=n;
              b=y;
            }
        void Print ( )
            { cout << a << b;
            }
};
```

```
int main()
{ abc m;
  m.set(10);          // m.a = 10
  m.print();          // 10 will be printed

  xyz n;
  n.set(20,30);       // n.a=20, n.b=30;
/* it is the case of function overloading.
   now n.a=20 is possible bcoz 'a' in class abc
is declared as protected, hence can be accessed by
the objects of the its immidiate derived class 'xyz'
within its member functions. */

  n.Print();
/* it will print 20 and 30. it is also the case of function
overloading. */
```

=> typecasting in inheritance

```
class abc
    { protected:
      int a;
    public:
      void Set_a(int n)
        { a=n;
        }
    friend void Print(abc);
    };

class nyt: public abc
    { int b;
    public:
      void Set_ab(int n, int y)
        { a=x;
          b=y;
        }
    };

void Print(abc m)
  {
    cout << m.a;
  }
main()
    { abc k;
      k.Set_a(10);   // k.a = 10

      Print(K);      // it will print 10 bcz
                        m.a = k.a = 10
```

xyz n;
n.set_ab(20,30); // n.a = 20, n.b = 30

Print(n); // 20 will be printed

// it will print 20 even if argument to be passed to the print() is an object of class abc. It actually does typecasting internally because 'n' is an object of class xyz which is derived from the base class 'abc'.

Date
Page

Constructors and destructor in inheritance

=) A constructor of derived class must first call a Constructor of base class to construct the base class instance of the derived class.

=) In order to invoke a parameterized Constructor of the base class, the constructor of derived class does an explicit call but in case of calling a default cons^{r} of the base clan, an implicit call of the base clan cons^{r} will take place from the cons^{r} of the derived class.

=) when a destructor (dest^{r}) of derived class object is called, after that base class destructor is also called.

=) when a derived class cons^{r} is called, it first calls base class cons^{r} implicitly or explicitly. So, technically, base class cons^{r} finishes its execu^{n} first and then derived class cons^{r} finishes its execu^{n}. So, derived class dest^{r} will finish its execu^{n} 1st, then base class dest^{r} will finish its execu^{n}.

```
class abc
{ protected:
  int a;
  Public:
    abc()
      { a=10;
      }
    abc(int x)
      { a=x;
      }
  ~abc(){ cout << "base destructor";}
```

```
class xyz
    { int b;
    public:
        xyz ( int x, int y)
        { b= 20
          abc ( x);
          b= y;
        }
       xyz ( )
        {
          b= 20;
        }
      ~xyz()
        {
          cout << "derived destructor";
        }
    };

int main( )
   {
      abc m;
/* it will invoke abc( ) & set m.a= 10

      xyz n(4,5);
/* it will invoke xyz (int, int) which in turn explicitly invokes
   abc (int). abc(int) will set n.a = 4
and finally, xyz(int, int) will set n.b=5 */

      xyz k;
/* it invoke xyz ( ) which in turn implicitly invokes abc()
abc() will set k.a=20 & finally xyz( ) will set k.b=20.
```

At the end derived class destr will be called b/c base derived class constr finished its execn after the base class class constr. Hence, following will be printed after the end of the program.

derived class destructor
base class destructor