OOPS

unit :- 2

class is a collection of data and function. The data or ~~class~~ variable which is declare inside the class is called member variable. The function which declare inside the class is called member function.

```
class rectangle.
{
Private:
    int len, br;
Public:
    void get_data();
    void area();
    void print_data();
};
```

```
Class class-name
{
Private:
    ~~sta~~ variable declaratio;
        function declaratio;
Public
    ~~vo~~ ~~class~~ variable decl;
        funct decle;
};
```

Class object :-

Object is used to access the member of a class. Any number of object can be created for the same class

```
class ~~St~~ student s1.

int s1.rollno;
```

```
class student *s1
{
int    s1 -> rollno.
```

Class scope

Method of accessing class member.
The class has two types of member
① Data Member
② Member function

Data member declared as private
Member function declared as public.

Data member of a class are accessed are used by member function of a class.

Member function accessed outside the class scope resolution will be used.

Example:-

```
#include <iostream.h>
using namespace std;
class student
{
Private:
int rollno;
~~int char~~
char name[20];

Public:
void get_data();
void display_data();
};

void student::get_data()
{
cout<<" Enter roll no: ";
cin >> roll no;
cout <<" Enter name: ";
cin >> name;
}

Void student:: display_data()
{
cout <<"\n Roll no= "<<rno;
cout <<"\n name ="<<name
}

int main()
{
Student s1, s2;
s1. get_data();
s1. display_data();
s2. get_data();
s2. display_data();
return;
}
```

Friend function :-

It is a non member function it is used to access private data of the class

Syntax:

friend datatype friend function (class1 , class2);

(or)

(argument class)

-> The friend function is a function that is not a member function of the class but it can be access the private and protected members of the class.

friend function is a given by keyword friend.

These are special functions which are declared anywhere in the class but have given special permission to access the private member of the class.

Adding two numbers:-

```
#include <iostream.h>
Class second;
Class first
{
Private:
  int a;
Public:
  Void read_A()
  {
   a=10;
  }
  friend int Sum (first f, second s);
};
```

```
class second.
{
Private:
  int b;
Public:
  void read_B()
  {
   b=20;
  }
friend int sum(first f, seconds
};
```

```
int sum(first f, second s)
{
  return f.a + s.b;
}
int main()
{
    first obj1;
    second obj2;
    obj1.read_A();
    obj2.read_B();
Cout << "sum is" << sum(obj1, obj2);
  return 0;
}
```

Multiplication

```
#include <iostream>
using name space std;
Class x;
Class Y
{
Private:
 int Y;
Public:
Void multiplication(Xa, Yb);
  Y()
  {
  y=20;
  }
};
Class X
 Private:
   int x;
friend void Y::multiplication(Xa,Yb);
Public:
  X()
  {
  x=10;
```

```
friend void Y::multiplication(Xa, Yb)
{
  int result;
  result = a.x * b.y;
  Cout << "In Multiplicat of a
        20 = " << result;
}
int main()
{
X Xa obj;
Y Yb obj;
Yobj.multiplication(Xobj, Yobj);
}
```

Constructor

The constructor is a specialized function with the name same as that of the class name. The constructor is a function used for initialization of data members.
It can be declared in public mode.
It can invoked automatically when an object gets created.
The constructor should not have any return type.
Even void type should not be written for the constructor.

Types of constructor

1. Default constructor
2. Parameterized constructor
3. Copy constructor

Default constructor:
constructor without argument or parameters

The default constructor is a function having the same name as its class name. This function has no parameters passed to it. Its purpose is to simply initialize the data members of the class.

```
#include <iostream.h>
class Sample
{
private:
  int a,b;
```

```
Public:
  void sample();
    {
    a = 10;
    b = 20;
    }
    void display()
    {
    cout << "\n a = " << a << "\n b = " << b;
    };

    int main()
    sample obj1, obj2;
    obj1.display();
    obj2.display();
    return 0;
    }
```

All the object contain same value.

② Parameterized Constructor

Ans The parameterized constructor is a function having same name as its class name and some parameters are passed to it.

Eg.

```
#include <iostream.h>
class num
{
private: int a,b;

  Public:
   num (int i, int j)
   {
   a=i;
   b=j;
   }
   void display ()
   {
cout << "\n a = " << a << "\n b = " << b;
```

```
int main()
num n1(10,20);
num n2 = num(30,40);
n1.display();
n2.display();
}
```

Every object has its own value.

Copy Constructor :-

→ creating a new object with old object

→ Copy Constructor is a special type of constructor in which new object is created as a copy of existing object. In which one object is initializing by other object

General for :-

```
Classname (classname & object)
{
// duty constructor
}
```

Syntax :-

```
classname new_object name (old object name);
```

Ex :

```
#include <iostream.h>
class num
{
Private:
  int a;
Public:
   num (int i)
   {
    a=i;
   }
   num (num &obj)
   {
    a =obj.a;
   }
void display()
{
cout <<"\n a:" << a ;
};
int main()
{
 num obj1 (10);
 num obj2 (obj1);
 obj2. display();
}
```

Destructor

The destructor is called when the object is destroyed automatically when the scope of the object end (i.e when the function end) or explicitly by using operator delete.

The destructor must have the same name as the class, and precedes with a tilde sign (~) and it must also return no value.

we normally use the destructor when the object occupies the dynamic memory in its lifetime.

```
#include <iostream.h>
class ~~image~~ sample
{
private:
    ~~int *height, *width~~
    static int count;
    Public:
        sample ()
        {
        count++;
    cout << "\n No. of objects created so far:" << count;
        }
    ~ Sample ()
        {
        count--;
cout << "\n no. of object remaining after destroyed" << count;
        }
    };
    sample :: count;
    main()
     sample obj1, obj2, obj3;
     return 0;
    }
```

Abstraction

Data abstraction is one of the characteristic of C++ by which simplified representation of the C++ program is presented to the user.
The implementation details are hidden.
Class is achieve data abstraction in C++.

Exe:-

```
class student
{
 private:
     int roll;
     char name [10];
 Public:
     void input ();
     void display();
};
```

the main function, we can access the functionalities using the object.

t instance -

```
tudent obj;
bj . input();
bj . display ();
```

om the above code implementation detail hidden.

UNIT:- 3

Inheritance

Inheritance is a property by which we can derived or create a new class from a base class.

One of the key benefits of inheritance is to minimize the amount of duplicate code in an application by sharing common code among several subclasses.

→ Reusability: The base class code can be used by derived class without any need to rewrite the code.

→ Extensibility: The base class logic can be extended in the derived class.

→ Data hiding: Base class can decide to keep some data private so that it cannot be altered by derived class.

→ overriding: With inheritance, we will be able to override the methods of the base class so that meaningful implementation of the base class method can be designed in the derived class.

→

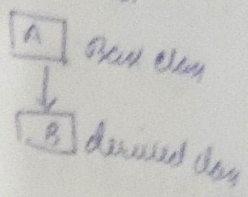
Types of Inheritance :-

1. Single inheritance
2. Multilevel inheritance
3. Multiple inheritance
4. Hierarchical inheritance
5. Hybrid inheritance

① Single inheritance :

When only one class is derived from a single base class then it is called single inheritance.

Ex:

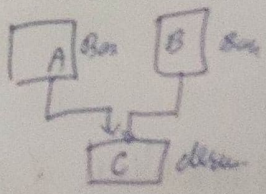



```
#include<iostream.h>
class A
{
 Public:
  int a;
 };

 Class B: Public A
  {
   Public
    int b;
    B()
    {
      a = 10;
      b = 20;

    void sum()
    {
    cout <<"\n sum is " <<(a+b);
    }
  };
```

```
int main ()
{
 B obj;
 obj.sum();
 return 0;
 }
```

② Multiple Inheritance :

The process of creating a class from multiple ~~inheritance~~ base class is known as multiple inheritance



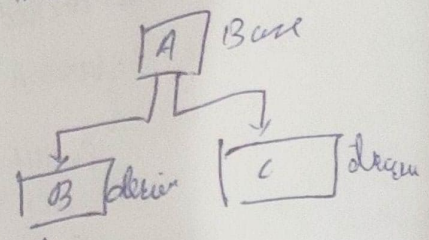

```
#include<iostream.h>
class A
{
Public:
   int a;
 };
 class B
  {
  Public:
    int b;
    };
```

```
Class c: Public A, Public B
  {
  Public:
    int c;
    c()
   {
   a = 1;
   b = 2;
   c = 3;
  }
```

```
Void sum()
{
cout <<"\n sum is" <<a+b+c;
 }
};
int main ()
{
c obj;
obj.sum();
return 0;
}
```

③ Hierarichal Inheritance:

→ The process of creating several derived class from a single base class is known as ~~info~~ hieracly inheritance

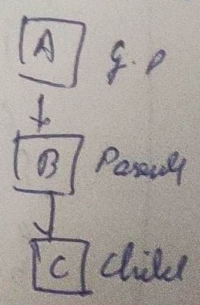



```
#include <iostream.h>
 Class A
 {
  Public:
   int a;
   };
 Class B: Public A
  {
  Public:
     int b; B()
     {
      a=1;
      b=2;
     }
     void sum()
     {
     cout <<"\n sum is << (a+b);
     }
 };
```

```
Class C: Public A
  {
   Public:
      int c; c()
      {
      a=1;
      c=3;
      }
      void mum ()
      {
      cout <<"\n multiplo is << (a*c);
      }
   };

int main()
B obj1;
obj.sum ();
C obj2;
obj.Mum ();
return 0;
}
```

④ Multilevel Inheritance:-

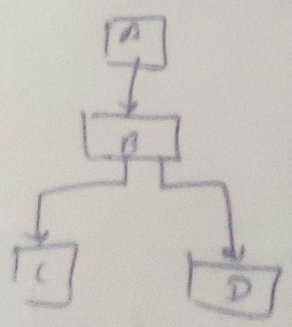
The process of creating a class from a derived class is know as multilevel inheritance.

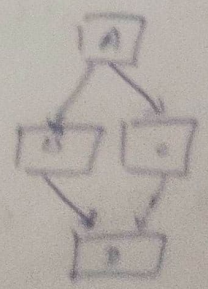


```
#include <iostream.h>
class A
{
 Public: int a;
};
class B: Public A
{
 Public: int b;
};
class C: Public B
{
 Public: int c;
 C()
 {
  a=1;
  b=2;
  c=3;
 }
void mul()
{
cout<<"\n mul is" <<m(a*b*c);
}
};
int main()
{
 C obj;
 obj.mul();
};
```

⑤ Hybrid Inheritance :

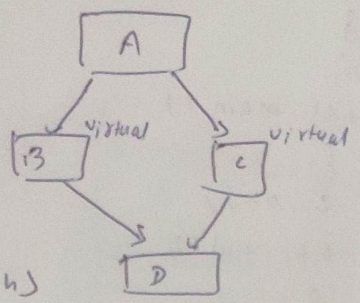
Hybrid inheritance is a combination of hierarchical and multilevel inheritance





Virtual Base Class

* Multipath inheritance or virtual base class is a combination of Multilevel inh. + Multiple inh. + hierarchy inh.



By win virtual the so only one copy of derived class form.

```
# include <iostream.h>
Class A
{
Public : int a;
};
Class B : Public virtual A
{
Public : int b;
};
Class C : Public virtual A
{
Public : int c;
};

Class D : Public B , Public C
{
~~Public~~ : int d;
D()
{
a = 1;
b = 2;
c = 3;
d = 4;
}
Void Sum()
{
Cout << "\n Sum is = " << (a+b+c+d);
};
```

```
int Main()
{
D obj;
obj.sum()
return 0;
}
```

In order to prevent the computer from giving an error due to ambiguity in multiple path or multiple inheritance, we use the keyword virtual.

That means when we inherit from the base class in both derived class, the base class is made virtual.

☆ Static And Dynamic Binding

① Static binding / early binding / compile time polymor

This is a type of binding in which the function call is associated with the function definition at the compile time only.

```
class Base
{
Public:
void display ()
{ cout <<"\n In base class" << endl; }
};
Class derived : Public Base
{
Public:
    void display ()
  { cout <<"\n In derived class" << endl; }
};
int main ()
{
Base b;
Derived d;
b.display();
d.display();
return 0;
}
```

SYLLABUS

Page No.
Date

→ late binding / Dynamic binding / runtime binding

This type of binding is called the late binding because the compiler cannot resolve the call function to appropriate function late until the runtime. late binding is acheived using virtual function

```
class Base
{
Public:
virtual void display()
{cout <<"\n In base class" << endl; }
};
```

```
Class derived: Public Base
{
Public:
void display()
{cout <<"\n In derived class" << endl;
};
```

```
int main()
{
Base *ptr;
Base b;
Derived d;
ptr = &b;
ptr -> display();
ptr = &d;
ptr -> display();
return 0;
}.
```

Virtual function:-

```
#include <iostream.h>
Class Base
{
Public:
  virtual void display ( )
   {
   cout <<"\n virtual display ( ) in base class";
   }
   void show()
   {
   cout <<"\n Non virtual show in base class";
   }
};
class Derived : Public Base
{
 Public
   void display ( )
   {
   cout <<"\n virtual displayes in derived class";
   }
   void show ( )
   {
   cout << " \n Non virtual show() in derived class";
   }
};
int main ()
{
  base * bptr;
  base bobj;
  bptr = &bobj;
  cout <<"\n Base class pointer shows
        to base class object";
  bptr -> display();
  bptr -> show();
  derived dobj;
  bptr = &dobj;
  cout <<"\n Base class pointer shows
        to derived class object";
  bptr -> display();
  bptr -> show();
  return 0;
}.
```

-> A virtual function is a member function that is declared with a base class and redefined by a derived class."

-> The virtual keyword is preceded to the function name.

-> The virtual function can be redefined in the ~~de~~ derived class. Thus using virtual function we can have one interface, multiple functions performing different task. This feature is called polymorphism.

-> The virtual function within the base class defines the form of the interface to that function. Each redefinition of the virtual function by a derived class implies some different task related to derived class

SYLLABUS

Page No.
Date

Pure virtual function

→ A pure virtual function is a virtual function which is to be implemented by derived class. It contain only declaration not definition.

→ The class that contain the pure virtual function is called abstract class.

→ The pure virtual function are declared using pure specifier i.e = 0.

```
#include <iostream.h>
Class A
{
Public:
virtual void display() = 0; // Pure virtual function
};

Class B: Public A
{
Public:
Void display()
{
Cout <<"\n In derived class B"<<endl;
}
};

Void main()
{
A* p;
B b;
C c;
p = &b;
p->display();
p = &c;
p->display();
}

Classc: Public A
{
Public:
void display()
{
Cout << "\n In derived class c"<<endl;
}
};
```

Virtual Destructors

→ virtual destructor is basically a base class destructor preceded by the keyword. virtual. The base class is declared as virtual in order to deallocate the memory of derived class explicitly. For example.

```
class Base
{
Public:
Base()
{ cout <<"\n Calling Base class constructor"; }

virtual ~Base ()
{ cout <<"\n Calling base class destructor"; }

};
class Derived : Public Base
{
Public:
Derived()
{ cout <<"\n Calling derived class constructor"; }

~ Derived ()
{ cout <<"\n Calling derived class Destructor"; }

};

int main()
{
Base* obj = new derived ();
delete obj;
return 0;
}
```