Date: 20-6-18
MON TUE WED THU FRI SAT

Chapter-1

Introduction & Features

1 Procedure Oriented Programming

→ High level languages such as Cobol, fortran and C are known as Procedure Oriented Programming (POP). Primary focus is on function.

fig. (Typical Structure of

→ It basically consist of writting a list of instruction for the computer to follow and organizing this instructions into groups known as function.

→ In multi function program many important data items are placed as global. So they can be accessed by all function.

→ Each function has that local data.

Characteristics of POP

→ Emphasis on doing the things.

→ Large programs are divided into Smaller programs known as functions.

→ Most of the functions share global data.

→ Data move openly around the System from function to function.

→ Function transform data from one form to another.

Date:
MON TUE WED THU FRI SAT

→ It employees top down approach in program design.

2 Object Oriented Programming

→ OOP breads data as a critical element in the program development and does not allow it to flow freely around the system

→ OOP allows decomposition of a problem into a number of entities called objects and then builds data and function around these objects

Features of OOP

→ Emphasis is on data rather than procedure.
→ Programs are divided into objects
→ Data Structures are designed such that they characterize the objects.
→ Function that operates all the data of an object are tied together in the data structure.
→ Data is hidden and cannot be accessed by external function
→ Objects may communicate with each other through functions,
→ New data & function can easily added whenever necessary

→ It follows bottom up apporach in program design.

Write a program to find average of two numbers.

```
#include <iostream>
using Namespace Std;
  int main()
{  int a, b, Sum=0, avg;
   cout << "a";
   cin >> a;
   cout << "b";
   cin >> b;
   Sum = a+b;
   avg = Sum/2;

   cout << "avg is " << avg;
}
```

Basic Concepts of OOP
object
class
Data Abstraction
Encapsulation
Inheritance
Polymorphism
Data binding
Message Passing

Date:
MON TUE WED THU FRI SAT

1. object

Object are the runtime entities in the object oriented system. They may represent a person, a place, a bank account or any item that the program has to handle.

object Student
Data Name
DOB
Marks
Function Total
Avg
Display

2. Class

The entire set of data and code of an object can be made a userdefined datatype with the class.

objects are variable of type class. A ~~task~~ Class is thus a collection of objects of similar types.

Fruit mango
Person Student
Vehicle Car

3. Data Encapsulation

The wrapping of data and function into a single unit called

Date.:
MON TUE WED THU FRI SAT

class is known as encapsulation. The data is not accessible to outside world and only those functions which are wrapped in the class can access it.
This insulation of the data from direct access by the program is known as data hiding or information hiding.

4. Data Abstraction

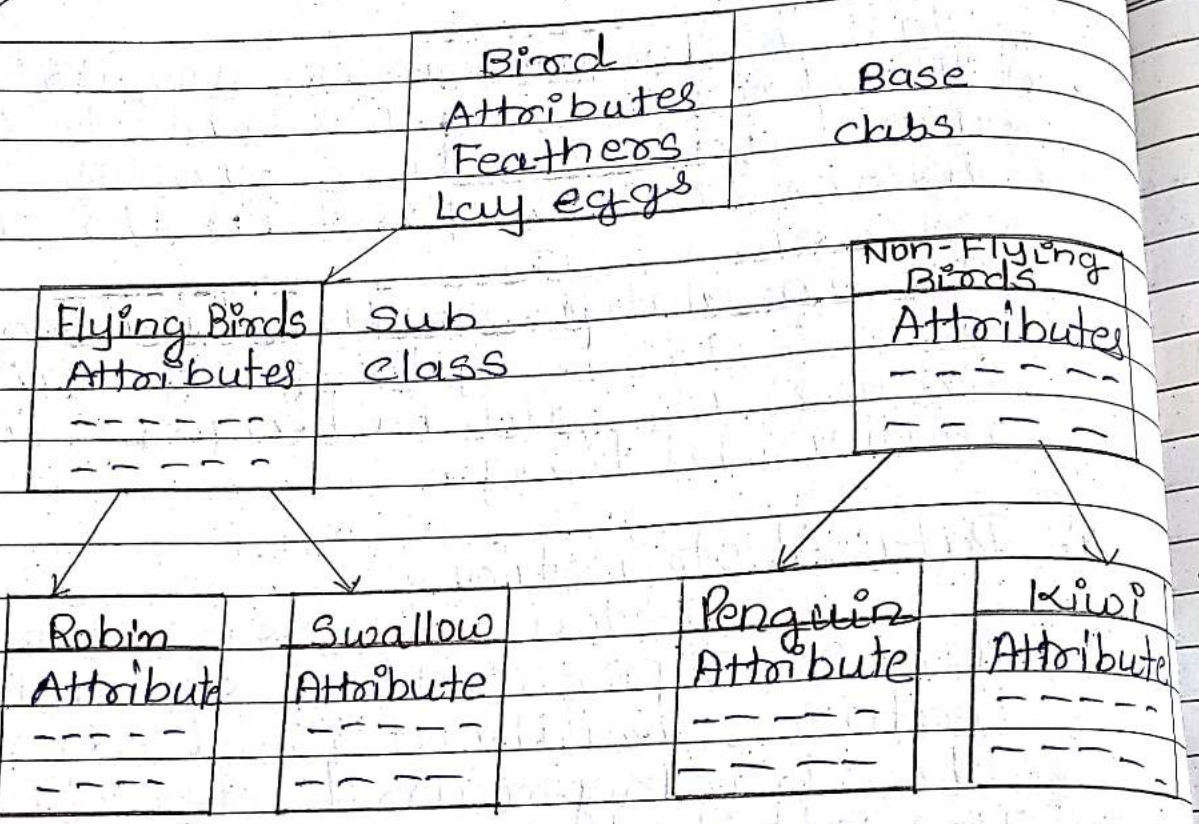
Abstraction refers to the act of representing essential features without including the background details. The attributes are sometimes called "data members" because they hold information. The function that operate on this data are known as methods or "member function".

5. Inheritance

Inheritance is a process by which object of one class can acquire the properties of objects of another class. It supports the concept of "hierarchical classification".

The advantage of inheritance is it provides reusability.

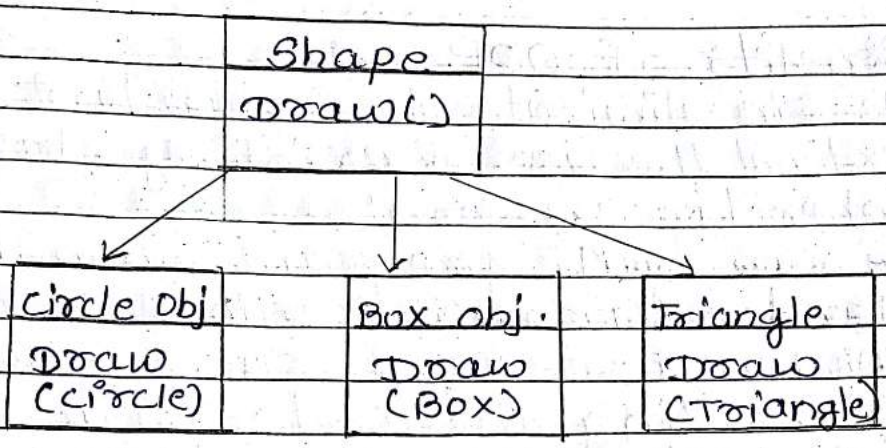
Date.: MON TUE WED THU FRI SAT



6. Polymorphism

It is the Concept in which an operation may exhibit different behaviour in different instances. The process of making an operator to exihibit different behaviour in different instances operator overloading. Using a single function to perform different type of class is known as function overloading

Date:
MON TUE WED THU FRI SAT



polymorphism

7. Dynamic Binding

It is also known as late binding means that the code associated with a given procedure called is not known until the time known as runtime. It is associated with polymorphism and inheritance.

8. Message Passing

An OOP consist of a set of object that communicate with each other it envolves three steps

i) Creating classes that define objects and their behaviour

ii) Creating objects from class defination

or

iii) Establishing Communication between objects.

eg: employee.Salary(amount);

↑ object ↑ message ↑ information

Date.:
MON TUE WED THU FRI SAT

Benefits of OOP

- We can eliminate the redundant code & extend the use of existing classes using inheritance.
- We can built program from the standard working modules then communicate with each other, instead of writting the code from scratch. This helps in saving development time & gives higher productivity.
- The principles of data hiding helps the programmer to build secure program. It is possible to have multiple instances of an object to coexist without any interference.
- It is possible to map objects in the problem domain to those in the program.
- It is easy to partition the work in a project based on object.
- The data center design approach enables to capture more details of a model in an implimentation form.
- Object oriented system can be easily upgraded from small to large system.
- Message passing technique for communication bet^n object makes the interface simple.
- Software complexity is easily managed.

Code - simple, reusable

Date.:
MON TUE WED THU FRI SAT

Application of OOP

- Real time system → weather forecasting, live television
- Simulation & Modeling
- Object oriented database
- Hypertext, Hypermedia
- Artificial Intelligence
- Neural Network & Parallel Programming
- Decision Support & office automation System
- CAD System (Computer Aided Design)
  - ↳ civil eng.

Date: 2/7/18
MON ☑ TUE ☐ WED ☐ THU ☐ FRI ☐ SAT ☐

Unit-3

CLASS, OBJECT AND FUNCTIONS

Introduction to class & object

The only difference between a structure and a class in c++ is that by default a member of a class are private whereas the member of a structure are public by default

a) Class Declaration
b) Class function Definition

→ A class is a way to bind the data and its associated functions together.
→ It allows the data and functions to be hidden if necessary from external use.
→ A class specification has two parts.

→ Class declaration describes the type & scope of its member. The class function defination describes how the class function are implimented.

Class Declaration

```
class class_name
{
    Private:
        variable declaration;
        function declaration;
```

Data.:
MON TUE WED THU FRI SAT

```
public :
    variable declaration;
    function declaration;
};
```

example :-

The keyword private & public are known as visibilites / labels.

```
class item
{
    int number;
    float cost;

public:
    void getdata(int number, float cost);
    void printdata();
};
```

By default members of class are private.

Creating object

```
Class item
{

} item1, item;
```

Date:
MON TUE WED THU FRI SAT

- Accessing class member

object_name.functionname (actual argument);

eg: item1.getdata(123,100.00);
item1.putdata();

eg: class XYZ
{
int x;
int y;
public:
int z;
}; xyz
int P;
P.x = 0; //⇒error as x is private mem.
P.z=10; //✓ as z is public data mem.

- Defining member function

Member functions can be defined in two places
i) outside the class definition
ii) inside the class definition.

i) ⇒ SYNTAX:
return-type class-name::function-name (parameters)
{
function body;
}

Date.:
MON TUE WED THU FRI SAT

eg: Void item :: getdata (int a, float b)
{
number = a;
cost = b;
}

Void item :: putdata (void)
{
Cout << "Number" << number;
Cout << "Cost" << cost;
},

eg: Class item
{
int number;
float cost;
public:
Void getdata (int a, float b);
void putdata ();
} item1;

Member function have the foll. characteristics

i) Different classes can use same function name

ii) Member function can access the private data of the class except friend function

iii) A member function can call another member function directly without using dot (.) operator.

ii) ⇒ SYNTAX :

```
eg : class item
     {
        int number;
        float cost;
      public :
        void getdata (int a, float b) ; //Dec
        Void putdata(void)
     {
        Cout << number ; // Definition inside
        Cout << cost ;      the class
     }
};
```

Example program for class implementation on

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

class item
{
    int number;
    float cost;
  public :
    void getdata (int a, float b); // Prototype
                                      declaration
    void putdata (void) // Function declared
{                          inside class
    Cout << "number" << number;
    cout << "cost" << cost;
```

```
    }
};
//member function definition
Void item :: getdata (int a, float b) // Use
{                                      membership labels
    number = a;
    cost = b;
}

int main()
{
    item x; //object
cout << "object x";
x.getdata (100, 299.95);
x.putdata();

item y;
cout << "object y";
x.q
y.getdata (200, 175.50);
y.putdata();
return 0;
}
```

```
object x
number : 100
cost : 299.95
object y
number : 200
cost : 175.50
```

Date.:
MON TUE WED THU FRI SAT

This program features the class item. This class contains two private variables & two public function. The member function getdata() which has been defined outside the class supplies value to both the variables. The member functions can have direct access to the private data items
E.g → number = a;
The member function putdata() is defined inside the class & behaves like & inline function. This fun displays the value of the private variables 'number' & 'cost'.
The program creates two objects 'x' & 'y'.

Making an outside function inline

SYNTAX

```
class item
{
    ----
    ----
    public:
    void getdata (int a, float b);
};
inline void item :: getdata (int a, float b)
{
    number = a;
    cost = b;
}
```

Write a C++ program to enter record of Student. Define a class student which contains foll. details.

Name of student
EN. No. of student
Contact No.

Member function
to get details of student
to display details of student

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

class student
{
    char name[25];
    int e_no;
    int contact_no[10];

  public:
  void getdata();
  void display();
};
student
class::
void getdata()
{
    int i;
    cout<<"Enter Your name"
        <<endl;
    cin>>name[i];
```

```
    cout << "Enter your enrollment
        no." << endl;
    cin >> e_no;
    cout << "Enter Contact no." <<
                endl;
    cin >> Contact_no[];
}
void Student :: display()
{
    cout << "Name" << name
            << endl;
    cout << "Enroll no" << e_no
            << endl;
    cout << "Contact no" <<
            Contact_no << endl;
}

Void main()
{
    Student S1;
    cout << "Student S1";
    S1. getdata();
    S1. display();
}
```

Write a C++ program of Bank System to insert the details of 5 m. Define a class Bank System which contains foll.

i) Data member
   a) name of account holder
   b) account no
   c) Balance
ii) Member
   a) To get details of acc.
   b) To display details of acc.
   c) To search acc. by acc. no.

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

Class Bank
{
    char name[25];
    int acc_no[5];
    in float balance;

  public:
    Void getdata();
    Void display();
    Void search();
};

Void Bank::getdata()
{
```

```
        int i;
    for (i=0; i<5; i++)
    {
        cout << "Enter Name" << S[i].name
                            << endl;
        cin >> S[i].name;
        cout << "Enter Acc. No" << endl;
        cin >> S[i].acc_no;
        cout << "Enter Balance" << endl;
        cin >> S[i].balance;
    }
}
Void Bank :: display()
 {
        int i;
    for(i=0; i<5; i++)
 {
    cout << "Name :" << S[i].name;
    cout << "Acc. No :" << S[i].acc_no;
    cout << "Balance :" << S[i].balance;
  }
}

Void Bank :: search()
{
    int i, b

    cout << "Enter the Acc. no to
                    Search" << endl;
    cin >> b;
```

Date.:
MON TUE WED THU FRI SAT

```
for (i=0; i<5; i++)
{
    if ( b== S[i].acc_no)
    {
        S[i].display();
    }
}
}

int main()
{
    Bank S
    S.getdata()
      display()
      Search()
  return 0;
}
```

Write a C++ program that demonstrate how private member function can be accessed of a class.

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

class employee
{
    int no;
    char name[20];
    float sal;

  void putdata()
{
   cout<<endl<<"Emp no"<<no;
   cout<<endl<<"Emp name"<<name;
   cout<<endl<<"Emp sal"<<sal;
}
  public:
void getdata()
{
   cout<<"Enter no";
   cin>>no;
   cout<<"Enter name";
   cin>>name;
   cout<<"Enter salary";
   cin>>sal;
   putdata();
};
```

Date: 16-7-18
MON TUE WED THU FRI SAT

```
int main()
{
    employee e1;
    e1.getdata();
// e1.putdata(); is illegal, object can't access
    return 0;    private class data.
}
```

Arrays with a class.

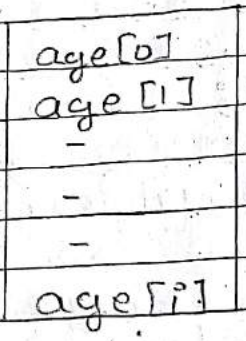
eg:- class person

```
{
    int age[10];
    public:
    void getdata();
    void pushdata();
};p;

int main()
{
    P.getdata();
    P.pushdata();
    return 0;
}
void person :: getdata()
{
    int i;
  for (i=0; i<10; i++)
  {
    cin >> age[i];
  }
}
```

Date: 16-7-18
MON TUE WED THU FRI SAT

Storage of Array of objects :-



Array of objects

```
class person
{
    int age;
    char name[10];
    public:
        void getdata();
        void pushdata();
} P[10];

int main()
{
    int i;
{
    int i;
  for(i=0; i<10; i++)
  {
      P[i].getdata();
      P[i].pushdata();
  }
    return 0;
}
```

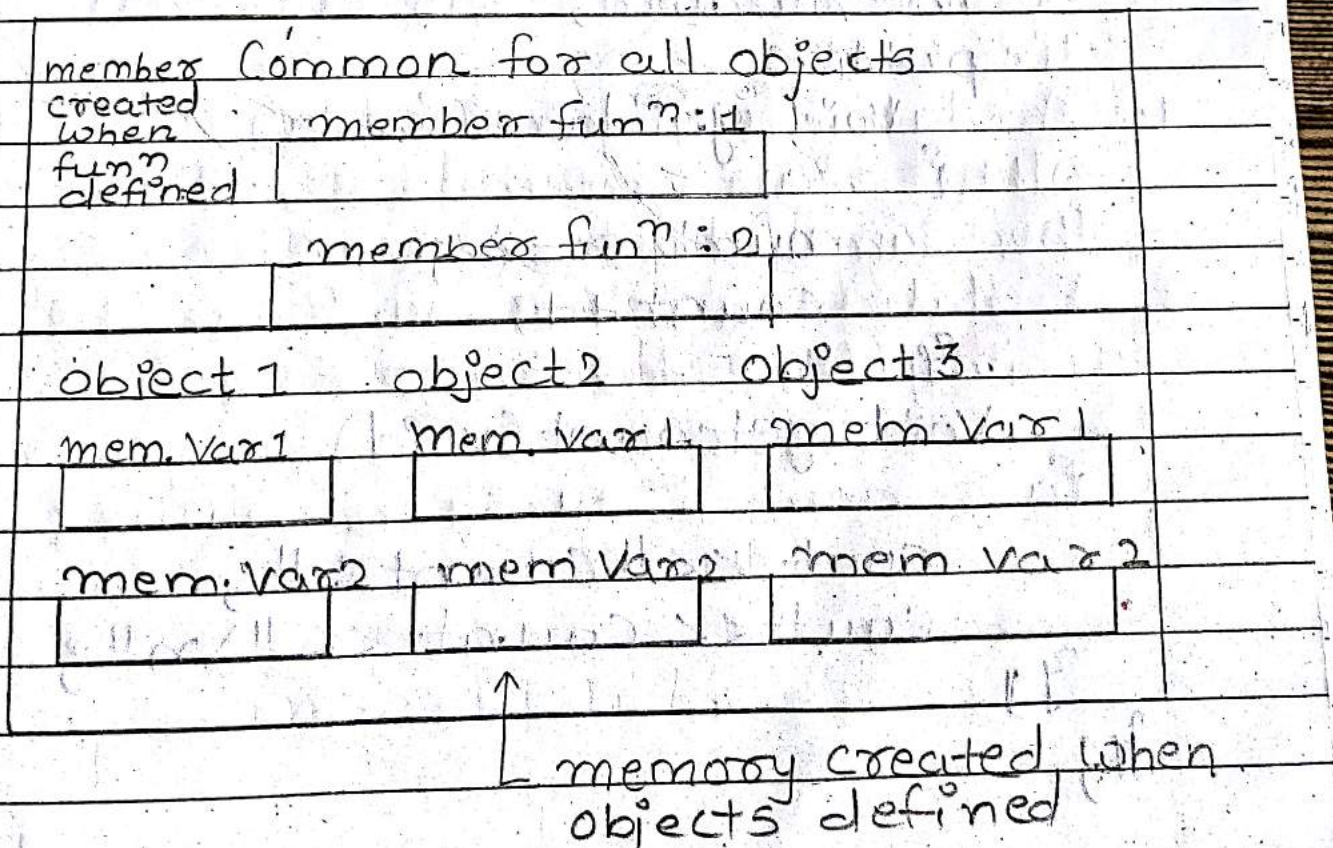
Storage of object

age	} p[0]
name	
age	} p[1]
name	
-	} p[i]
-	

Memory Allocations for Objects

- The memory space for object is allocated when they are declared and not within the class specified.
- space for member variable is allocated separately for each object.

Fig: object of memory



Date:
MON TUE WED THU FRI SAT

Static Data Members

A static member Variable has foll. characteristics

1. It is initialized to zero. when the first object of its class is created.
2. Only one copy of that member is created for the entire class and shared by the all the objects of that class.
3. It is visible only within the class but its lifetime is in entire program.

```
#include <iostream>
using namespace std;
class item
{
    static int count;
    int number;
    public:
      void getdata (int a)
      {
          number = a;
          count++;
      }
    void getcount (void)
{
    cout << "count : ";
    cout << count << "\n";
}
};
```

```
int item::count;    // define
int main()
{
    item a,b,c;
    a.getcount();
    b.getcount();
    c.getcount();

    a.getdata(100);
    b.getdata(200);
    c.getdata(300);

    cout<<"After reading data:"<<\n";
    a.getcount();
    b.getcount();
    c.getcount();

    return 0;
}
```

Output :-

```
Count : 0
Count : 0
Count : 0
After reading data:
Count:3
Count:3
Count:3
```

Since the data is read into objects three times the variable count is incremented three times. Because

there is only one copy of count shared by all three objects.

Static variables are like non-inline member functions as they are declared in a class declaration & defined in the source file.

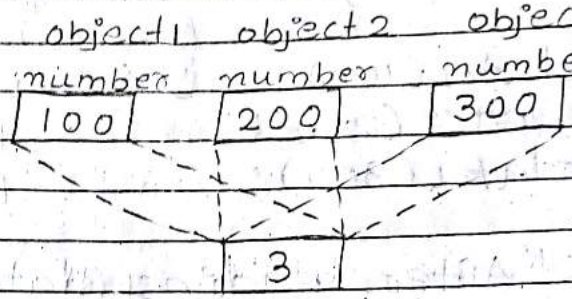


fig:- sharing of static data member

Static member functions

A member function that is declared static has the foll. properties

1. A static fun^{n} can have access to only other static members declared in the same class.
2. A static member fun^{n} can be called using the class name as follows

class-name :: function-name;

eg:

Date.:
MON ☑ TUE ☐ WED ☐ THU ☐ FRI ☐ SAT ☐

eg:

```
#include <iostream>
using namespace std;
class test
{
    int code;
    static int count;
    public:
    void setcode(void)
    {
        code = ++count;
    }
    void showcode(void)
    {
        cout << "object number:" << code <<
                                    "\n";
    }
    static void show count(void)
    {
        cout << "cout :" << count << "\n";
    }
};
int test::count;
int main()
{
    test t1, t2;
    t1.setcode();
    t2.setcode();
    test::showcount();
    test t3;
    t3.setcode();
```

```
test::Showcount();
t1.Showcode();
t2.Showcode();
t3.Showcode();
return 0;
}
```

Output

count: 2
count: 3
object number: 1
object number: 2
object number: 3

The static fun^n showcount displays the number of objects created till that moment.

→ A count of number of objects created is maintained by static variable count.

→ The statement (code = ++count) is executed whenever setcode fun^n is invoked and the current value of count is assigned to code.

→ Since, each object has its own copy of code the value contained in code represents a unique number of its object.

e.g

Date.:
MON TUE WED THU FRI SAT

* Array Of Objects (Done)

pg-110 (program for array of object)

Objects As Function Argument

An object may be used as a function argument like any data type. It can be done in two ways

1. A copy of entire object is passed to the fun". (Pass by value)
2. Only the address of the object is transfered to the fun". (Pass by refe.)

e.g

```
#include <iostream>
using namespace std;
class time
{
    int hours;
    int minutes;
    public:
void gettime(int h, int m)
{
    hours = h;
    minutes = m;
}
  Void puttime(void)
{
    Cout << hours << "\n";
    Cout << minutes;
}
```

Date.:
MON TUE WED THU FRI SAT

```
void Sum (time, time);
{};

void time :: Sum (time t1, time t2)
                                //define
{
  minutes = t1.minutes + t2.minutes;
  hours = minutes / 60;
  minutes = minutes % 60;
  hours = hours + t1.hours + t2.hours;
}

int main()
{
    time T1, T2, T3;
    T1.gettime (2, 45);
    T2.gettime (3, 30);
    T3.Sum (T1, T2);
    Cout << T1.Puttime ();
    Coub << T2.puttime ();
    Cout << T3.Puttime ();
    return 0;
}
```

Friendly Function

The function definition doesn't use the keyword friend or Scope resolution operator.

A friend fun^n although not a member fun^n has full access right to the private members of the class.

Date.:
MON TUE WED THU FRI SAT

3

A friend fun^n has special characteristics as follows

1) It is not in the scope of the class to which it has been declared as friend.
2) Since, it is not in the scope of class, it can't be called using object of that class.
3) It can be called like a normal fun^n without the help of any objects.
4) It cannot access the member names directly & has to used an object name & (.) membership operator with each member name.
5) It can be declared in either public or private part of the class.
6) It often uses object as argument.

e.g :

```
#include <iostream>
using namespace std;
class result2;
Class result1;
{
    private:
        int no_of_sub;
    public:
        int getdata()
    {
        cout << "Enter no. of Subjects";
        cin >> no_of_sub;
    }
```

```
friend int display(result1 r1, result2 r2);
};
class result2
{
    private:
        int total_marks[3];
    public:
        int getdata()
    {
        cout<<"Enter the marks";
        cin>>total_marks;
    } friend int display(result1 r1, result2 r2);
};      int display(result1 r1, result2 r2)
    {
     return(r2.total_marks/r1.no_of_sub);
    }
        int main()
{
    result1 r1;
    result2 r2;
    r1.getdata();
    r2.getdata();
    cout<<"avg="<<display(r1,r2);
    return 0;
}
```

Date.:
MON TUE WED THU FRI SAT

5

Constructor

A Constructor is a special member function of a class whose task is to initialize the object of its class. It is special because it's name is same as the class name. It is invoked whenever the object of its associated class is created.

eg:

```
class integer
{
    int m,n;
    Public:
    integer();
};
integer::integer()
{
    m=0;
    n=20;
}
```

If the above class their is a Constructor declared. In order to invoke the Constructor of class integer the foll. object declaration statement can be made:

```
integer intI;
```

Date.:
MON TUE WED THU FRI SAT

Rules

A Constructor that accepts no parameter is called default Constructor.

Characteristics of a Constructor.

1. They do not have return types not even void. and therefore they cannot return any data.
2. They should be declared in public section of the class.
3. They are invoked automatically when the object is created.
4. They cannot be inherited, though a derived class can call a base class. constructor.
5. Like other C++ fun^n, they can have default argument.
6. They can not be virtual.
7. We cannot refer to their address.
8. An object with a Constructor cannot be used as a member of a union.
9. They make implicit calls to the operat new & delete when memory allocation is required.

Ex:-

Date.:
MON TUE WED THU FRI SAT

Ex:

```
class student
{   int i;
    int m;
    char a[10];

  Student (int x, char b[10]);
  void putdata()
  {
      Cout << "Your Roll no" << m;
      Cout << "Your name" << a[i];
  }
};
  Student :: Student (int x, char b[10])
{   int i;
    m = 16;
    a[i] = Snehal;
}
int main()
{
    Student S;
    S. putdata();
  return 0;
}
```

Parameterized Constructor

The Constructor that can take arguments is called as parameterized constructor.

Date.:
MON TUE WED THU FRI SAT

```
Class integer
{
    int m, n;
    public:
    integer (int x, int y);
};
integer :: integer (int x, int y)
{
    m = x;
    n = y;
}
```

A parameterized constructor can be called implicitly or explicitly by making the foll. object declaration statement.

```
integer int1 (4, 50); // implicit call
integer int1 = integer (4, 30) // explicit call
```

Const Member Function

If a member fun^n does not alter any data in the class then we may declare it as const member fun^n.

Eg: `void mul(int, int) const;`

Date.:
MON TUE WED THU FRI SAT

	Member fun^n	Constructor
1)	It's name shouldn't be the same as the class name.	It's name should be same as class name.
2)	It has to be called explicitly.	It is called implicitly at the time of creating object.
3)	It can be called any no. of times on object.	It is called only once on objects.
4)	It's operation is modifing & accessing.	It's operation is initialization.
5)	They are inherited.	They are not inherited.

```
#include<iostream>
using namespace std;
class TIME
{
    int hour, minute, second;
    public:
     TIME()
    {
      hour=0; minute=0; second=0;
    }
     TIME(int a, int b, int c)
    {
      hour=a;
      minute=b;
      second=c;
```

```
};
void sum(TIME t1, TIME t2)
{
    second = t1.second + t2.second;
    minute = second/60;
    minute=
    second = second % 60;
    minute = minute + t1.minute +
                t2.minute;
    hour = minute/60;
    minute = minute % 60;
    hour = hour + t1.hour + t2.hour;
};
void display()
{
    cout << hour << ":" << minute
         << ":" << second;
};
};
int main()
{
    TIME t;
TIME t1(10, 30, 25);
TIME t2(0, 30, 20);
TIME t3;
  t3.sum(t1, t2);
  t3.display();
  return 0;
}
```

Scanned by CamScanner

```
1. Cout << "x=" x;
2. m=5; //n=10; //m+n=5;
3. Cin >> x; >> y;
4. Cout << \n " Name : " << name;
5. Cout << " Enter Value : " << E; Cin >> x;
6. /* Addition */ z = x+y;
```

Multiple Constructors in a class.

ex:

```
class integer
{
    int m, n;
    public:
1)  integer()
    {
        m=0; n=0;
    }
2)  integer(int a, int b)
    {
        m=a;
        n=b;
    }
3)  integer (integer & i)
    {
        m=i.m;
        n=i.n;
    }
};
```

Date.:
MON TUE WED THU FRI SAT

```
#include <iostream>
using namespace std;
class complex
{
    float x, y;
    public:
    complex()
{ }
    complex ( float a)
    {
        x = y = a;
    }
    complex (complex, complex)
                float real, float imag;
{
    x = real;
    y = imag;
}
friend complex sum (complex, complex);
friend void show (complex);
};

complex sum (complex c1, complex c2)
{
    complex c3;
    c3.x = c1.x + c2.x;
    c3.y = c1.y + c2.y;
    return (c3);
}
```

Date.:
MON TUE WED THU FRI SAT

```
void show(complex c)
{
    cout << c.x << "+j" << c.y << "\n";
}

int main()
{
    complex A(2.7, 3.5);
    complex B(1.6);
    complex c;
     C = sum(A, B);
    cout << "A=" ; show(A);
    cout << "B="; show(B);
    cout << "C="; show(C);

complex P, Q, R;
P = complex(2.5, 3.9);
Q = complex(1.6, 2.5);
R = sum(P, Q);
cout << "P=" ; show(P);
cout << "Q=" ; show(Q);
cout << "R=" ; show(R);
cout << "\n";
return 0;
}
```

```
A = 2.7 +j 3.5
B = 1.6 +j 1.6
C = 4.3 + j 5.1
P = 2.5 +j 3.9
Q = 1.6 +j 2.5
R = 4.1 +j 6.4
```

Date.:
MON TUE WED THU FRI SAT

Dynamic Initialization of objects

objects can be initialized dynamically and the advantage is that we can provide various initialization formats using overloaded Constructors.

```
#include <iostream>
using namespace std;
Class fixed_deposit
{
    long int P_amount;
    int Year;
    float Rate;
    float R_Value;
public:
    Fixed_deposit()
    { }
Fixed_deposit(long int p, int y, float
                        r= 0.12 );
Fixed_deposit (long int p, int y, int r);
Void display (void);
};
Fixed_deposit :: Fixed_deposit (long int
                p, int y, float r)
{
    P_amount = P;
    Years = Y;
    Rate = r
    R_Value = P_amount;
```

```
for (int i=1; i<=Y; i++)
{
    R_Value = R_Value * (1.0+r);
}
}
Fixed_deposit::Fixed_deposit (long int P,
                    int Y, int r)
{
    P_amount=P;
    Years=Y;
    Rate=r;
    R_value=P_amount;

  for (int i=1; i<=Y; i++)
  {
    R_value = R_value * (1.0+ float (r)/100);
  }
}

Void Fixed_deposit::display (Void).
{
  Cout << "\n"
       << "Principal Amount=" << P_amount
<< "\n" << "Return Value=" << R_value
  << "\n";
}
```

```
int main()
{
    Fixed_deposit FD1, FD2, FD3;
    long int P;
    int Y, R;
    float r;
Cout << "Enter amount, period, interest
  rate (in percent)" << "\n";
Cin >> P >> Y >> R;
FD1 = Fixed_deposit (P, Y, R);

Cout << "Enter amount, period, interest
  rate (decimal form)" << "\n";
Cin >> P >> Y >> r;
FD2 = Fixed_deposit (P, Y, r);

Cout << "Enter amount and period"
                    << "\n";
Cin >> P >> Y;
FD3 = Fixed_deposit (P, Y);
Cout << "\n Deposit 1";
  FD1.display();
}
```

Same for FD2.display FD3.display

Output:

```
Enter amount, period, interesent rate
   (in percent)
 10000  3  18
Enter amount, period, interesent rate
  (decimal form)
10000  3  0.18
```

Date.:
MON TUE WED THU FRI SAT

Enter amount and period
10000 3
Deposit 1
principal Amount = 10000
Return value = 16430.3
Deposit 2
principal Amount = 10000
Return value = 16430.3
Deposit 3
principal Amount = 10000
Return value = 14049.3

Copy Constructor

A Copy Constructor is used to declare and intialize an object from another object.

Ex: integer I2 (I1);
integer I2 = I1;

The process of initializing through a Copy Constructor is known as. Copy initialization.

A Copy constructor takes a refoences to an object of the same class. as itself as an argument.

Date.:
MON TUE WED THU FRI SAT

```
#include<iostream>
using namespace std;
class code
{
    int id;
    public:
    code() { }
    code(int a) { id=a; }
    code(code &x)
    {
        id= x.id;
    }
    void display(void)
    {
        cout << id;
    }
};
int main()
{
    code A(100);
    code B(A);
    code C=A;
    code D;
    D=A;
    cout<<"\n id of A:";
        A.display();
    cout<<"\n id of B:";
        B.display();
    cout<<"\n id of C:";
        C.display();
```

```
cout <<"\n id of D: ";
   D.display();
  return 0;
}
```

output
```
id of A:
id of B:
id of C:
id of D:
```

Destructor

A Destructor is used to destroy the objects that have been created by a constructor.

Ex: ~ abc() { }

A destructor never takes any argument nor does it return any value. It is a good practice to declare destructors in a program since it releases the memory space for future use.

```
#include <iostream>
using namespace std;
int Count = 0;
Class test
{
    public:
      test()
    {
      Count++
      Cout <<"\n\n Constructor msg";
```

```
    object number "
  << Count << " Created... ";
}
  ~test()
{
  cout << "\n\n Destructor msg:
object number " << count << "destro-
    yed";
  Count--;
}
};
int main()
{
  cout << " Inside the main block ";
  cout << " Creating first object ";
  test T1;                    //Block 1
{
cout << "\n Inside Block 1";
Cout << "\n Creating two more objects
    T2 and T3";
  test T2, T3;
Cout << "\n Leaving Block 1";

}
cout << " Back to main Black;
  return 0;
}
```

Output :

Inside the main block
Creating first object
Constructor msg : object number 1
Created...
Inside the Block 1
Creating two more objects T2 & T3
Constructor msg : object number 2
Constructor msg : object number 3
Created...
Leaving Block 1
Back to main Block
Destructor msg : object number 3
destroyed
Destructor msg : object number 2
destroyed
Back to main Block
Destructor msg : object number 1
destroyed

Operator Overloading :

Mechanism of giving special meaning to an Operator is known as operator overloading. We can overload all C++ operator except : Class member access operator (., .*), Scope resolution operator (::), size operator (sizeof), Conditional operator (? :)

Date.:
MON TUE WED THU FRI SAT

Defining Operator Overloading

return type classname :: operator op(arglist)
{
  Function body // task defined
}

process of operator overloading includes three step.

1) Create a class that define the datatype that is to be used in overloading operator.
2) Declare the operator function 'operator op' in public section of the class. It may be either a friend fun^n or member fun^n.
3) Define the operator function to implement a required operation.

Overloaded operator fun^n can be invoked by ① op x ② x op
For unitary op. ① x op y
For Binary operator

Overloading Unitary Operator

#usi

```
Class Space
{
    int x, y, z;
    public:
```

```
    void getdata(int a, int b, int c);
    void display(void);
    void operator-();
};
void space::getdata(int a, int b, int c)
{
    x=a;
    y=b;
    z=c;
}
void space::display(void)
{
    cout<<"x="<<"\n";<<x;
    cout<<"y="<<"\n";<<y;
    cout<<"z="<<"\n";<<z;
}
void space::operator-()
{
    x=-x;
    y=-y;
    z=-z;
}
int main()
{
    space s;
    s.getdata(10,-20,30);
    cout<<"s:";
    s.display();
    -s;
    cout<<"-s:";
    s.display();
    return 0;
}
```

Date.:
MON TUE WED THU FRI SAT

Output

S: 10, -20, 30

-70

36

S: x = 10
y = -20
z = 30

-S: x = -10
y = 20
z = -30

Rules for Overloading Operator

→ Only existing operators can be overloaded.

→ The overloaded operator must have at least one operand that is of user defined type.

→ Overloaded operator follows the syntax rule of the original operator

→ There are some operators that cannot be overloaded

→ We cannot use friend function to overload certain operation.

→ However member fun^n can be used to overload this operators.

* Ex: "=", '()', '[]', '->' cannot be overloaded using friend function

→ Unary operators overloaded by member fun^n do not take explicit argument & do not return argument.

→ Binary operators overloaded through member function takes one explicit argument and those which are overloaded using friend fun^n takes two explicit argument.

→ When using binary operator overloaded to member fun^n, left hand operand must be an object of relevant class.

→ Binary arithmetic operators must explicitly return a value.

Overload Prefix And Postfix Operators.

```
#include<iostream>
using namespace std;
class check{}
{
   private:
    int i;
   public:
    check()
{
      i=0;
}
void operator ++()
{
      ++i;
}
void display()
{
 cout<<"i= "<<i<<endl;
```

```
};
int main()
{
    Check obj;
    obj.display();
    ++obj;
    obj.display();
    return 0;
}
```

Overloading instream & outstream operator.

```
#include <iostream.h>
using namespace std;
class Distance
{
 private:
    int feet;
    int inches;
public:
   Distance()
{
   feet = a;
   inches = 0;
}
   Distance (int f, int i)
{
    feet = f;
    inches = i;
```

```
};
friend ostream &operator<<(ostream & output, const Distance &D)
{
    output << "F" << D.feet << "I" << D.inches;
    return output;
};
friend istream &operator>>(istream & input, Distance & D)
{
    input >> D.feet >> D.inches;
    return input;
};
};
int main()
{
    Distance D1(1,2), D2(2,3), D3;
    Cout << "Enter value of obj 3";
    Cin >> D3;
    Cout << "1st" << D1;
    Cout << "2nd" << D2;
    Cout << "3rd" << D3;
    return 0;
}
```

NOTE > It is imp. to make operation operator overloading fun^n, a friend of class because it could be called without creating an object.

Date.:
MON TUE WED THU FRI SAT

Overloading Binary '+' operator

```
#include<iostream.h>
using namespace std;
class Complex
{
   float x;
   float Y;
   public:
     complex ()
   {
   }
      Complex (float real, float imag)
{
        x = real;
        y = imag;
}
  Complex operator + (complex)
{
     Complex temp;
     temp.x = x + c.x;
     temp.Y = y + c.y;
      return temp;
}
void display ()
{
     cout << x << y;
}
};
```

```
int main()
{
    Complex C1, C2, C3;
    C1 = Complex(2.5, 3.5);
    C2 = Complex(1.6, 2.7);
    C3 = C1 + C2;
    Cout << "C1=" ; C1.display();
    Cout << "C2=" ; C2.display();
    Cout << "C3=" ; C3.display();
    return 0;
}
```

Pg → 158 (Diagram)
→ 159 (program)

The statement $C_3 = C_1 + C_2$ invokes operators '+' function.

We know that a member function can be invoked only by an object of the same class. Here, the object C_1 takes the responsibility of invoking the fun^n & C_2 plays the roles of an argument that is passed to the fun^n. Therefore the data members of C_1 are accessed directly & the data members of C_2 are accessed using the (.) dot operator.

As a rule in overloading of binary operators the left hand operand is used to invoke the operator fun^n. & the right hand operand is passed as an argument.

Date.:
MON TUE WED THU FRI SAT

Unit : 5
Pointers, Virtual Functions And Polymorphism

Pointer : Pointer is a derived data type that refers to another data variable by storing the variables memory address rather than data.
- Pointers provide an alternative approach to access other data objects.

SYNTAX :
data-type * pointer-name ;
- A variable must be initialized before using it in a c++ program.

eg :- int *ptr, a ; // declaration
ptr = &a ; // initialization

```
#include <iostream.h>
#include <conio.h>

Void main()
{
  int a, * ptr1, ** ptr2 ;
  clrscr() ;
  ptr1 = & a ;
  ptr1 = & ptr1 ;
Cout << "address of a: " << ptr1 << "\n" ;
Cout << "address of ptr1: " << ptr2 << "\n" ;
Cout << "After incrementing the address
          values: " ;
  ptr1 += 2 ;
```

Date.:
MON TUE WED THU FRI SAT

```
Cout<<" The address of a :<<ptr1<<"\n";
ptr2 += 2;
Cout<<" The address of ptr1 :"<<ptr2
                    <<"\n";
}
```

OUTPUT:

address of a : 0x8fb6fff4
address of ptr1 : 0x8fb6fff2
After incrementing the address values:
The address of a : 0x8fb6fff8
The address of ptr1 : 0x8fb6fff6

We can also use Void pointers it is also known as generic pointer, which refers to variables of any datatype. The pointers which are not initialized in a program are called null pointers.

Manipulation of a Pointer (Dereferencing)

We can manipulate a pointer with the indirection operater which is also known as derefrence operator. with this operator we can indirectly access the data variable contained.

SYNTAX

```
* pointer variable
```

Dereferencing a pointer allows us to get the contain of the memory location that the pointer points to. using it we can also change the contain

address Hexa/octal

Date.:
MON TUE WED THU FRI SAT

of the memory location.

1024	a = 20
1025	

```
int main()
{
    int a=10;
    int *p;
    p=&a;
    Cout<<"The Value of a is:"<<*p;
    *p=*p+a;
    Cout<<"New Value of a :"<<a;
    return 0;
}
```

& → address of
* → value of

```
float b=10.5;
int a=10;
int *p;
× p=&b;
✓ p=&a;
```

Output
The Value of a is: 10
New Value of a : 20

*p-=11 → Short hand operator

*p = *p - 11
= 10 - 11
= -1

Date:
MON TUE WED THU FRI SAT

```
int a[5];
int *p;
p = &a[0];
p++; → 1026 a[1]
p = p+2; → a[3] = 1026 + 2
// p+=2;
p--; a[1] = 1026
```

Pointers with array

```
Void main ()
{
    int a[10] = {1, 9, 4, 37, 8, 3, 19, 4, 5, 6, 57};
    int *p, i;
    p = a; ⇒ p = &a[0] (same)
    Cout << "Value at pointer : " << *p;
    p++;
    Cout << "Value at pointer : " << *p;
    p--;
    Cout << "Value at pointer : " << *p;
    p+=2;
    Cout << "Value of p+2 : " << *p;
    p = p-1;
    Cout << "Value of p-1 " << *p;
    p+=5;
    Cout << "value of p+5 " << *p;
}
```

Output:

Value at pointer : 1
Value at pointer : 9
Value at pointer : 1

Date:
MON TUE WED THU FRI SAT

Value of P+2 : 9
Value of P-1 : 1
Value of P+5 : 37

// Ex program for manipulation of pointer

Arrays of Pointer

An array of pointers point to an array of data items. Each element of the pointer array points to an item of the data array. Data items can be access either directly or by dereferencing the elements of the pointer array

```
int *iarray[10];

#include <iostream.h>
#include <conio.h>
#include <string.h>
#include <ctype.h>

Void main()
{
    int i=0;
    char *ptr[10]={"book", "television",
        "Computer", "sports"};
    char str[25];
    clrscr();
    cout<<"Enter your favourite leisure
        pursuit:";
```

Date.:
MON TUE WED THU FRI SAT

```
cin >> str;
for (i=0; i<4; i++)
{
  if (!strcmp (str, *ptr[i]))
  {
    cout << " Your favourite pursuit "
         << " is available here" << endl;
    break;
  }
}

if (i == 4)
cout << " Your favourite pursuit is not
     available";
  getch();
}
```

Example program to search an elem. using pointer from an array

```
# include <iostream.h>
# include <conio.h>
Void main()
{
  int arr[10] = {1, 99, 4, 37, 88, 3, 19,
                 45, 62, 87};
  int i, num, *ptr;
  ptr = arr;
  cout << " Enter the element to be
       Searched ";
```

```
cin>> num;
for (i=0; i<10; i++)
{
  if (*ptr == num)
  {
    cout << "\n" << num << "'s present in array";
    break;
  }
  else if (i == 9)
  cout << "number is not present the array";
  ptr++;
}
getch();
}
```

A Pointer of String

```
char *ptr = "one";
char num[] = "one";
```

Pointers to Objects

A pointer can point to a object created by a class. object pointer's are useful in creating objects at run time.

Ex:

```
item x;
item *ptr = &x;
```

Here the pointer ptr is initialized with the address of x.
Member fun^n of class can be access in two ways
i) Using (.) dot operator & the object
ii) Using the arrow operater & the object pointer.
We can also create an array of objects using pointer.

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

Class item
{
    int code;
    float price;
    public:
        Void getdata(int a, float b)
        {
            code=a;
            price=b;
        }
        Void show()
        {
            Cout << "code : " << code;
            Cout << "price : " << price;
        }
};
```

Date:
MON TUE WED THU FRI SAT

```
int main()
{
    item obj;
    item *p;
    p = &obj;
    p->getdata(10,20);
    P->show();
    return 0;
}
```

The pointer p is initialized with address of object.

Array of Pointers to object.

Syntax: item *p[5];

To Create a array of five pointer to the object we use the above declaration. In Order to call any of the getdata fun^n of this pointer array fell. is the Statement

P[i] = getdata.(10,20);

where i points to any of the pointer index.

Date.:
MON TUE WED THU FRI SAT

This Pointer

C++ uses a unique keyword called this. To represent an object that invokes a member fun^n. This unique pointer is automatically pass to a member fun^n when it is called.

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

class one
{
   int a;
   public:
      one(int a)
   {  ->a = a
      a = x;
   }
   void show()
   {
      cout << a;
   }
};
int main()
{
   one ob;
   ob.show();
   return 0;
}
```

Date:
MON TUE WED THU FRI SAT

For example: One important application of this pointer is to return the object it points to

For ex: the statement return this inside a member function will return this inside a member fun the obj. that invoke the function
This statement assume the importance when we want to compare two or more obj inside a fun & return the invoking obj. as a result.

Pg-239

```
#include <iostream>
#include <cstring>
using namespace std;
class person
{
    char name[20];
    float age;
    public:
    person(char *s, float a)
{
    strcpy(name, s);
    age = a;
}
```

```
person &person :: greater (person &x)
{
    if (x.age >= age)
      return x;
    else
      return *this;
}
void display (void).
{
  cout << "Name : " << name;
  cout << "Age : " << age;
}
};
int main()
{
  Person P1 ("John", 39.50),
         P2 ("Ahmed", 29.0),
         P3 ("Hebber", 40.25);
Person P = P1. greater (P3);
Cout << "Elder Person is: \n";
   P. display();
   P = P1. greater (P2);
cout << "Elder Person is : \n";
   P. display();
   return 0;
}
```

Date.:
MON TUE WED THU FRI SAT

Pointer to Derived class

```
#include<iostream.h>
class BC
{
    public:
    int b;
    void show()
    {
        cout<<"b="<<b;
    }
};
class DC: public BC
{
    public:
      int d;
    void show()
    {
        cout<<"b="<<b
            <<"d="<<d;
    }
};
int main()
{
    BC *bptr;
    BC base;
    bptr= &base;
    bptr->b=100;
    cout<<"points to base";
    bptr->show();
```

```
DC derived;
bptr = &derived;
bptr -> b = 200;
cout << "points to derived";
bptr -> show();

DC *dptr;
dptr = &derived;
dptr -> d = 300;
cout << "dptr is derived pointer";
dptr -> show();
cout << "using ((DC*)bptr)";
((DC*)bptr) -> d = 400;
((DC*)bptr) -> show();
return 0;
}
```

OUTPUT : bptr points to base obj
b = 100
bptr now points to derived o
b = 200
dptr is derived Pointer
b = 200
d = 300
using ((DC*)bptr)
b = 200
d = 400

Date:
MON TUE WED THU FRI SAT

pointers to object of a base class are type compatible with pointers to objects of derived class i.e (if B is a base class & D is a derived class then a pointer declared as a pointer of B can also be a pointer to D
However their is a problem in using base pointer to access the public members of derived class.
It can be done using casting of base pointer to derived class type

Polymorphism

Polymorphism means one name having multiple forms

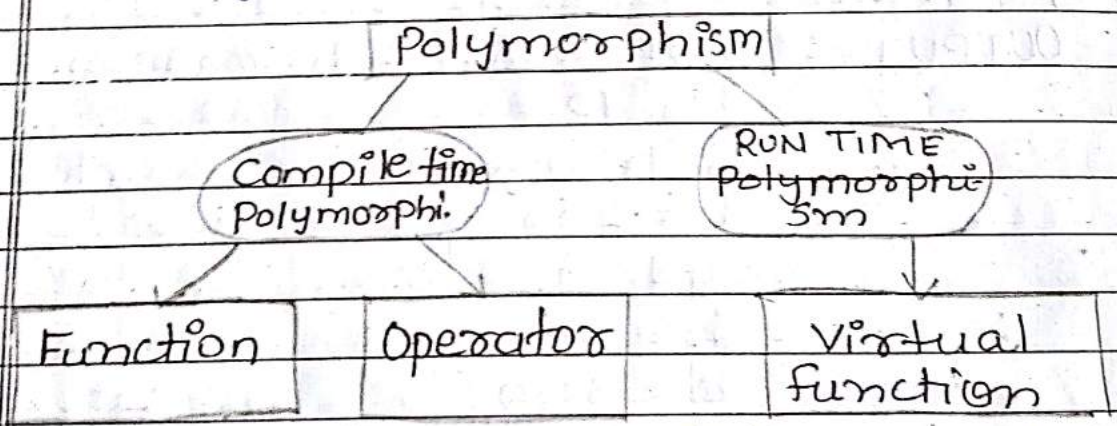


Fig: Acheiving Polymorphism

There are two type of polymorphism namely i) compile time ii) run time Functions & operator overloading are examples of Compile time polymorphism.
- The overloaded member fun^n are selected for involking by matching

arguments, both type and number. The compiler knows this information at the compile time & is able to select the appropriate fun^n for a particular call or compile time. This is called early or static binding.
It means object is bound to it's fun^n called at compile time
- In run time polymorphism an appropriate member fun^n is selected while the program is running done through virtual fun^n. It is called late or dynamic bindinding. Dynamic binding required use of pointers to object.

Virtual Function

When we have same fun^n name in both the base & derived classes the fun^n in the base class is declared as virtual using the keywoord virtual preceddiny its normal declaration.

When a fun^n is made virtual c++ determines which fun^n to use at run time based on the type of object pointed to by the base pointer.

Date.:
MON TUE WED THU FRI SAT

```
#include <iostream.h>
Class Base
{
    Public:
     Void display()
     {
      Cout << "Display base"; }
     Void virtual show()
     {
      cout << "show base"; }
};
Class Derived : public Base
{
    public:
  Void display()
 { Cout << "Display Derived"; }
  Void show()
 { Cout << "show Derived"; }
};
int main()
{
  Base B;
  Derived D;
  Base *bptr;
  cout << "1) bptr points to base";
  bptr = &B;
  bptr → display();
  bptr → show();
  cout << "bptr points to derived";
  bptr = &D;
  bptr → display();
```

```
    bptr -> show();
    return 0;
}
```

OUTPUT:
bptr points to base
Display base
show base
bptr points to Derived
Display base
show Derived

bptr -> display() calls only the fun^n associated with the base whereas bptr -> show() calls the derived fun^n version of show(). This is because the fun^n display has not been made virtual in the base class.

Pure Virtual function (Pg → 248)

A pure virtual function is a function declareing base class that has no definition relative to the base class. In such cases the compiler requires each derived class to either define the function or redeclare it as a pure virtual function.

Date.:
MON TUE WED THU FRI SAT

Virtual Constructor and Destructor

A Constructor cannot be virtual justify

1. object of the Constructor must be same as type of class which is not possible with a virtual Constructor
2. At the time of calling a Constructor the Virtual table would n't have been created to resolve any conflict to virtual fun^{n} calls.

```
Class A
{
    public :
    ~A()
    { // Base class destructor; }
};
Class B : Public A
{   public :
    ~B()
    { // Derived class destructor; }
};
main()
{
    A * ptr = new B()
    delete ptr ;
}
```