S.No.	Date	Title	Page No.	Teacher's Sign/ Remarks

Object Oriented Programming

C++

eg:
```
#include<iostream.h>
Using namespace std;
int main()
 {
  cout<< "Hello";
  return 0;
 }
```
(redirecting the output
cout: Printf)

.
```
#include<iostream.h>
Using namespace std;
int main()
 {
  int n1, n2;
  cin >> n1 >> n2;
  Sum = n1+n2;
  Cout << "Sum is" << sum;
  return 0;
 }
```

. Find max. of 3 numbers.

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

```
int main()
{ int a, b, c, max;
  Cout << "Enter 3 numbers";
  Cin >> a >> b >> c;
  max = a;
  if (b > a)
   {b > a
   { max = b;
   }
  else if (c > max)
   {
    max = c;
   }}
  cout << "max is" << max;
return 0;
}
```

Q. Find the sum of even and odd nos in a set of nos

```
#include <iostream>
using namespace std;
int main()
{
  int n, a, c, o, sum=0 SE, so=0; a;
  Cout << "How many nos?";
  Cin >> n;
  while{
  for{
  i=0 while (n<=0)
      {
      cout << "Enter number";
      cin >> a;
      if (a%2 == 0)
      SE+= a;
      else
       so+ = b;
      n--;
      }
  cout <<
```

PROCEDURAL ORIENTED PROGRAMMING VS OBJECT ORIENTED PROGRAMMING

. Characteristics of procedure oriented prog

- Emphasis is on doing things ⇒
⇒ Action oriented

2. Large programs are divided into smaller programs known as function.
3. Most of the functions share global data
4. Data moves openly around the system from function to function
5. Functions transform data from one form to another.
6. Controls top down approach to program design

OOP Characteristics

1. Emphasis on data rather than on procedure
2. Programs are divided into what are known as objects
3. Data structures are designed such that they can characterize the object ⇒ class
4. Functions that operate on the data are tied together in the data structure
5. Data is hidden and cannot be accessed by external functions
6. Objects may communicate with each other through functions.
7. New data and functions can be easily added whenever necessary. Easy maintenance
8. Follows bottom up approach in program design

Classes And Objects

Class : group of similar objects
showing same characteristics
+ common behaviour
+ shared relationship
DS (Data structure)

Eg: Class :- Flowers Plants
Objects :- Rose, Jasmine, Tulips, Sunflower, Marigold, Mango tree, Coconut tree

Characteristics: ~~Small~~ ~~color~~ ~~shapes~~ Flowers, ~~Herbs~~, Properties, ~~Shrubs~~, ~~trees~~, ~~climbers~~, ~~creepers~~, ~~roots~~, ~~leaves~~, Size, Phyllotaxy, Root system

Functions: ~~Medicinal purposes~~, flowering, germination, ~~Vegetation~~, ~~plant reproduction~~, photosynthesis, plant reproduction

Class: Computer

Objects: Toshiba, Dell, Apple, Predalon, Azus

Characteristics: ~~notepad~~, ~~Desktop~~ RAM, Processor, graphics pro

Functions: Programming, Debugging, process

CLASSES AND OBJECTS

Implementation

```
class <name of class>
{
  private : data
            membfn();
  Public : data
           member functions;
};
```

Access specifiers:
→ Private
→ Public

Function definition

1) Inside the class
2) Outside

1. Inside the class.

```
class item
  { Private: int code;
             float price;
             int qty;
    public: void show()
             { code=125;
               price = 200.55;
               qty = 100;
               cout <<code<< qty << price
             }
  };
```

2) Outside the class

```
class item
  { Private: int code;
             float price;
             int qty;
    Public: Void show();
  };
```

```
void item::show()
{ code = 100;
  price = 200.45
  qty = 500;
  cout << code << price << qty;
}
```

⇒
```
int main()
{
  item t1, *t;
  t1.code = 500;   X
  t1.show(), only public can be accessed.
  t -> show();
}
```

Q. Write a program using class to do two arithmetic operations using member functions.

```
#include<iostream>
using namespace std;
class arithmetic;
{ private: int a, b;
  private:- int read();
  public:- int read();
           int write();
           int add;
           int sub;
           int read;
           int quo;
};
int arithmetic:: read()
{
  cout << "enter two numbers:";
  cin >> a >> b;
}
int arithmetic:: add()
{ int s = 0;
  s = a + b;
}
int arithmetic:: sub()
{ int d = 0;
```

s = 0, d = 0, p = 1, q = 0;

```
    d= a-b;
    return d;
}
int arithmetic :: prod()
{ int p=0;
  p = a*b;
  return p;
}
float arithmetic :: quo()
{
  int q=0;
   q = a/b;
   return q;
}
int arithmetic :: write()
{
  cout<<"Sum = \n" <<s;
  cout<<"Difference = \n"<<d;
  cout<<"Product = \n" <<p;
  cout<<"Quotient = \n" <<q;
}
int main()
{
  arithmetic a1, a2;
        a1.read();
        a1.add();
        a1.sub();
        a1.prod();
        a1.quo();
        a1.write();
        return 0;
}
```

[7]2. WAP to read in int data member and display as follows.

Eg: 28 8 2018 ⇒ 28 Aug 2018

```
#include<iostream>
#include<string.h>
using namespace std;
class date
{
  Private: int d, m, y; Char a[10];
  Public: void read();
```

```
    Void write();
    Void month();
};
void date :: read()
{
Cout << "enter date in dd/mm/yy";
cin >> d >> m >> y;
}
    date :: month()
{
    if (m==1)
        strcpy(a, "January");
    else if (m==2)
        strcpy(a, "February");
    else if (m==3)
        strcpy(a, "March");
    else if (m==4)
        strcpy(a, "April");
    else if (m==6)
        strcpy(a, "June");
    else if (m==5)
        strcpy(a, "August");
    else if (m==10)
        strcpy(a, "October")
    else if (m==12)
        strcpy(a, "December");
    else
        strcpy(a, "Invalid");
}
int main()
{ Void Date:: write()
{
    cout << d;
    Puts(a);
    cout << y;
}
int main()
{
    Date d1, d2;
```

```
d1.read();
d1.month();
d1.write();
return 0;
}

//write()
{
if(d%10==1)
cout<<d<<"st";
else if(d%10==2)
cout<<d<<"nd";
else if(d%10==3)
cout<<d<<"rd";
else
cout<<d<<"th";
}
```

Properties Of OOPs

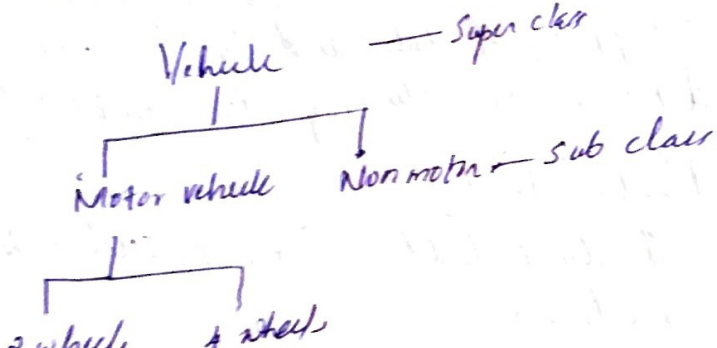
1. Encapsulation.
   related data is stored together as module.
   - wrapping of data and fn. in single module.
   data is accessible only to member functions.
2. Data hiding
   Data is hidden from outside ~~functions~~ parts.
   Access specifier: private, public, protected
   ∴ Pass by parameter can be avoided. member function can access

3. Abstraction.
   ↳ Member function
   Only necessary things are displayed
   Change the implementation
   → External interface is not changing
   But internal structure is changing

eg. sorting ⇒ : quick sorting, bubble sorting
obj.sort()

3. Inheritance
- beggest advantage - reusability
- Hierarchial structures

eg



No need to repeat old features again. Here 2 wheeler is a derived class of motor vehicle which inturn derived class of vehicle

4. Polymorphism
It is a way to redefine things.

eg: cout << i
<< - leftshift operator.
One single thing behaving differently in different situations
Operator overloading eg. <<
function overloading -

5. Dynamic Binding (Late time / run time binding)
When a function calls another, the relation between these 2 is called binding.
Compiler - static binding (Early)

6. Message Passing
~~How~~ objects interact with each other:
public member functions.

Q. Write a class which represents a triangle, the member functions should be
1. to check validity of a triangle
2. Display the sides
3. Find the area and display it.

```
Class Triangle
{ Private:
  int a, b, c;
  int area;
  int s;
  Public:
  void input ();
  int validety ();
```

```
void sides();
void area();
};
int huang::validity()
{
if (a+b)>c && (b+c)>a && (a+c)>b)
{ cout << " triangle is valid";
  return 1;
}
else
{
cout<< "triangle is invalid";
  return 0;
}
}
void huangle :: input()
{
cout << "Enter sides of triangle";
cin >> a >>b >>c;
}
void huangle :: sides()
{ cout << "the sides of huangles are " << a<<b<<c;
}
void huangle :: sides() area()
{
s = (a+b+c)/c;
area = sqrt(s*(s-a)*(s-b)*(s-c));
cout << " The area is " << area;
}
int main()
{
huangle t;
t. input();
Int a;
a = s. validity;
if (a==1)
{
s. area();
  s. sides();
}
else
  { cout << " Triangle is invalid"; }
}
```

C++ REFERENCE VARIABLES

→ Parameter passing techniques

1. Call by value
2. Call by address
3. Call by reference

Another name given to an existing variable - reference variable

eg int i;
i = 5;

⇒ int & ref = i; & ⇒ reference

Whenever ref is declared, it should be initialized. No special memory is created for ref.

int *ptr;
ptr = &i; (& address operator)

Printing: cout << i << ref << *ptr;

- Array of ptrs possible whereas array of ref it is not possible

- DIFFERENCES BTW POINTERS & REFERENCES
- U cannot have null references, reference should always be connected to a valid piece of storage
- Once a reference is initialized to an object it cannot be changed to refer to another object whereas pointers can be pointed to another object at anytime
- A reference must be initialized when it is created. Pointers can be initialized at any time
- Pointers can iterate over the array using ++. ref variable cannot
- A pointer to a class or structure uses → operator to access its members whereas a reference uses . operator.
- A pointer needs to be dereferenced with * to access the memory location, it points to a reference can be used directly.

eg. Int main()
{
int i;
int & r = i;
int *ptr = &i;
i = 5;

cout << "value of i" << i ; 5

cout << "Value of r" << r ; 5

r = 25 ;

cout << "value of i" << i ; 25

cout << "Value of r" << r ; 25

cout << "value of " << *ptr ; 25

*ptr = 30

eg 2: Reference variable to structure.

```
struct demo
{ int a;
};
Int main()
{ int x=5, y=6;
  demo d;
  int *p;
  p = &x; ✓
  p = &y; ✓
  int &r = x;
  &r = y; X } reference cannot be changed
```

r = y; (x=6) changing the value

p = NOLL; ✓

p = &x;

& r = NULL; X

p++; Pointer points to next memory location.

r++; (x=7)

cout << &p << &x ; diff

Cout << &r << &x ; same

```
demo *dq = &d;
demo &qq = d;
```

q => a = 10; ✓

q . a = 10; X

qq -> a = 10; X

qq . a = 10; ✓

Cout << p ; Address of x

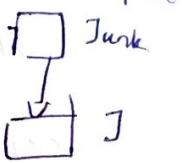
cout << x; 7 (value of x)

eg 3;

PARAMETER PASSING

```
Int main()
  {
   int junk = 4;
   cout << junk;
   f1(junk)
   cout << junk;  4
   f2(&junk); (call by address)
   Cout << junk; 5
   f3(junk);
   cout << junk; 6
```

```
Void f1(int J)
  { J++; }
Void f2(int *J)
  { (*J)++ }
```




```
Void f3(int &J)
  {
    J++;
  }
```

RETURN PARAMETER:

```
Int val() = {10,20,30,40,50};
Int & Setvalues(int i)
   { return val[i];
   }
Int main()
{
cout << "values before change";
for(int i=0;i<5;i++)
   cout << val[i];
 Setvalues(1) = 77;
 Setvalues(3) = 90
cout << "value after";
for(int i=0;i<5;i++)
{ cout << val(i);
}//:10,20,30,40,50
```

Output:

Value before change : ~~0 1 2 3 4~~

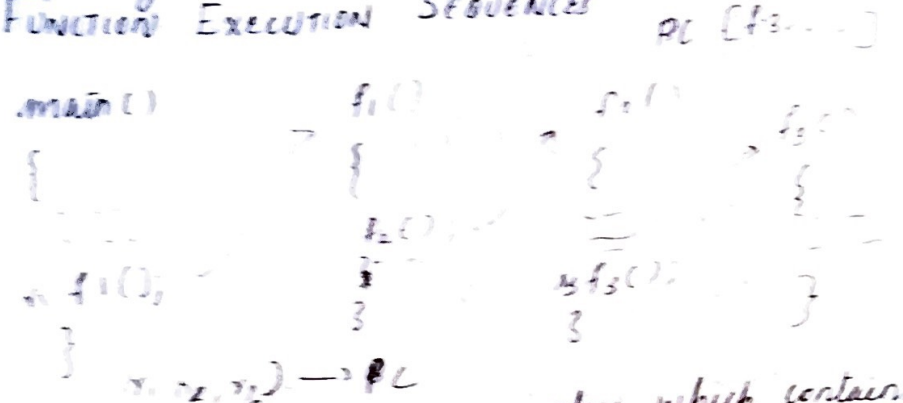
10 20 30 40 50

Values after change: 10 77 30 90 50

Inline Functions:

improving overhead time

- Function Execution Sequence



- Program Counter is a register which contains the address of next instruction to be executed.
- LIFO - Last In First Order - Stack

Overhead time

Saving of return address
Saving of registers
Saving of flags
} Retrieving

Inline fn definition

```
{ ---
  ---
}
```

→ function should be small
→ loop, switch, if else like branching statements should not be there
→ function shouldn't have static variables
→ function shouldn't be recursive

- Inline float cube(float a)

```
{ return (a*a*a);
}
int main()
{ float c = cube (3.5);
}
```

- Macros

#define sum 10 (set sum = 10)

Inline Functions Vs Macros

- #define SQUARE(v) v*v
- inline ~~float~~ square (~~float~~ int f)

```
{ return (f*f)
}
int main()
{ int - p=3, q=3, r, s;
```

```
S = SQUARE( ++a);
r = Square (r+p);
cout << " r = " << r << " s = " << s;
}
```

r = 25
s = 16

Function Overloading

- Type of parameter
- Number of the parameter
- Order of parameter
- Ret type is not a criteria
- Static/Nonstatic doesn't matter

~~int add (int, int)~~
~~float add float (float, float);~~

int add (int, int);
float add(float, float);
} Set of overloaded fn

float add (float, float, float)

~~Error list~~

typedef int count;
Sum (int);
Sum (count); Same

- Typedef declared types are not avoided
- Point to b array functions are not avoided

void modify (char *ptr)
Void ... (char a[])

Cases

1) Exact match
   d = add (5, 3)
   f = add (5.75, 3.33)

2) Matching through integral promotion
   char → int
   int → float
   float → double.

3) Matching through user defined functions conversions in the case of class.

→ Write overloaded functions of volume which returns volume of various shapes like cube, cylinder, sphere and cuboid

⇒
```
#include <iostream>
using namespace std;
int Volume (int a)
{
 return (a*a*a);
}
float volume (float r, int h)
{
 return (π*r*r*h);
}
float volume (float r)
{
 return (4/3*π*r*r*r);
}
int volume (int a, int b, int c)
{
 return (a,b,c);
}
Void main ()
{
 int a,b,c,
 float r,h;
 cout << "Enter your choice \n 1. Cube 2. Cylinder \n 3. Sphere \n 4. Cuboid \n";
 Cin >> 0;
 switch (0)
 {
 Case 1 : cout << "Enter side \n";
          cin >> a;
          Cout << Volume (a);
          break;
 Case 2 : cout << "enter radius and height \n";
          Cin >> r >> h;
          Cout << volume (r,h);
          break;
 Case 3 : cout << "Enter radius \n";
          cin >> r;
          cout << Volume (r)
          break.
 Case 4 : Cout << "Enter sides \n";
          cin >> a >> b >> c;
          cout << volume (a,b,c)
          break
 Default : cout << "Invalid"
 }
}
```

2. Write 3 versions of overloaded function sum

a) Takes an integer array and returns the sum of all elements of array

2) Takes an array and a character. If the character is E returns sum of even nos and if char is O returns sum of odd nos.

3) Passes array and 2 integers, swap the positions indicated by integers

```
#include <iostream>
using namespace std;
int sum( int a[], int n )
{
int s = 0; i ;
  for(i=0 ; i< n; i++)
{ s+= a[i] ;
  return s ;
}
}
int sum (int a[], int n, char b )
{
int s=0 , i , s1 = 0 ;
if (b=='E' || b=='e')
{ for(i=0 ; i<n; i++)
 { if (a[i] % 2 == 0)
   s+= a[i] ;
 }
return s ;
}
else if (b == 'O' || b == 'o')
{
 for(i=0 ; i <n ; i++)
 {
  if (a[i] % 2 != 0)
  s1 += a[i]
 }
 return s1 ;
}
int sum (int a[], int p, int q)
{
```

~~int b[2],~~ int g, i

```
for(i=0; i<n; i++)
    if((i+1) == x)
        a[i] = a[
```

```
int z;
z = a(p-1);
a[p-1] = a(q-1)
a(q-1) = z;
}
```

DEFAULT ARGUMENTS

```
int add(int, int);
int add(int, int, int);
int add(int, int, int, int);
```
} fn overloading

```
int add(int n1, int n2, int n3=0, int n4=0);
d = add(2,3)(n1=2, n2=3)
e = add(2,4,5)(" n3=5 )
```

f - add(1,2,3,4);

- Can be used for those arguments which rarely changes.
- Default values can be put on right end.

  d = (2, 3, , 5);x

- Assigning default values at breaking positions not possible

eg

```
int sum(int a, int b=10, int c=20, int d=30);
int main()
{
    int a=1, b=2, c=3, d=4, s;
    s = sum(a,b,c,d);   sum is 10
    cout << "sum is" << s;
    s = sum(a,b,c);     Sum is 36
    cout << "sum is " << s;
    s = sum(b,c);       55
    cout << "sum is " << s; 5 Sum is 55
    s = sum(d);
    cout << "sum is " << s; Sum is 64
```

```
s = sum (d,b,a);
cout <<"sum is"<<s ;
}
Int sum (int j, int k, int l, int m)
 {
 return (j+k+l+m);
 }
}
```

1+2+3+4 = 10

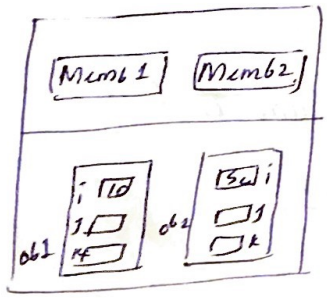
1+2+3+30 = 36

2+3+20+30 = 55

4+10+20+30 = 64

7+30 = 37

MEMORY ALLOCATION FOR OBJECTS:



ob1 . i = 6

ob2 . i = 50

How much memory : sizeof

```
Class data
 { int i;
  float j;
  char c;
 public : void setdata();
        {
         cin >> i >> j >> c
        }
        void display()
        {
        cout << i << j << c;
        }
};
int main()
{
 data d1, d2;
 cout << "size of d1" << sizeof (d1);
 cout <<"size of d2" << sizeof (d2);
}.
```

char = 1
int = 2
float = 4
double = 8

member functions do not belongs to object memory. Object memory contains space only for variables.

ARRAY OF OBJECTS

```
 int a[10];
```

data type for objects : class

item t1[10]; class name + array

t.[i].getdata());

STATIC MEMBERS OF A CLASS:

STATIC DATA:

Only one copy all objects.
- Change of 1 object is visible to other
- One time initialization.
(Initialized when the first object is created)

```
Class item
{ static int c; // Declaration
};
int item :: c=10;
```

eg Class number

```
{ static int c;
  int k;
  public: void zero()
          { k=0 }
          void count()
          {c++;
           k++;
           Cout<< "value of c"<<c
           Cout<<"value of k"<<k
};          }
```

```
int number :: c = 5;

int main()
{ number O1, O2, O3;
  O1.zero();
  O2.zero();
  O3.zero();
  O1.count();
  O2.count();
  O3.count
}
```

O/P:

	c = 6	k = 1
	c = 7	k = 1
	c = 8	k = 1

STATIC MEMBER FUNCTIONS

- It can access only other static members.
- Public static function, you can call them with · Class

```
Class item
{
  static void display()
    {
    }
};
```

call
item :: display();

```
Class beta
{
  Private :
     static int c;
  Public:
       static void count ()
       { c++; }
       static void display()
       { cout << " c = " << c; }
};
Int beta::c = 20
Int main()
{
    beta :: count();        c = 20
    beta :: display();      c = 21;
    beta obj;
    obj.count();
}
```

O/P:

Write a program to find results of set of students in one subject. The max and the pass mark for the subject are available as static data. The student is passed if he/she secure more than pass mark. Use a static member function for calculating and printing result.

```
#include <iostream>
using namespace std;
Class student
{
 Private: static int passmark = max;
 public:  static void calc( int P)
          { if (P > Passmark)
            cout << " Passed ";
            else
            cout << " Failed ";
          }
};  void read()
Int student::
          {
           cout << "Enter the mark of student ";
           cin >> m ;
          }
} a[10];
Int student:: max = 100 ;
```

```
int student:: Pass mark = 50;

int main()
{
    int n;
    student.read();
    student::calc();
}
cout << "Enter no. of students";
cin >> n;
for(i=0;i<n;i++)
{
    a[i].read();
    a[i]::calc();
}
```

PASSING OBJECTS AS FUNCTION ARGUMENTS:

```
Class
{
    int hrs;
    int mints;
    Public:
    Void get time (int h, int m)
    {
        hrs = h;
        mts = m;
    }
    void puttime()
    {
        cout << "hours " << hrs << "minutes " << mts;
    }
    Void sum(time t1, time t2)
    {
        mts = t1.mts + t2.mts;
        hrs = mts/60;
        mts = mts % 60;
        hrs = hrs + t1.hrs + t2.hrs.
    }
};
int main()
{
    time T1, T2, T3;
    T1.gettime(5,45);
    T2.gettime(3,30);
    T3.sum(T1,T2);
    T1.put time();
    T2.put time();
```

```
T3.put time ();
}
```

T3 = T2.sum(T1,T2); - return

~~time sum~~

~~void time~~

~~time sum (time t1, time t2)~~

~~{ hrs = t1.mts + t2.mts~~

```
time sum (time t1, time t2)
{ T2.no time T;
T2.mts = t1.mts + t2.mts;
T2.hrs = mts/60;
T2.mts = mts%60;
T2.hrs = hrs + t1.hrs + t2.hrs;
}
return T2;
};
```

T2.sum(T1);

```
Void sum (time t1, time t2)
{
T2.mts = t1.mts + t2.mts
T2.hrs = mts/60
T2.mts = mts%60
T2.hrs = hrs + t1.hrs + t2.hrs
}
```

CONSTANT MEMBER FUNCTION:

Its a member function which doesnt change variables in class.

```
Class A
{ int a;
  public: void add (int a, int b) const
          { c = a+b; X Error
            cout<<c; ✓ Printing allowed
            cout<<a+b; ✓
```

Used for printing purposes not for modification

WAP to find radius of a circle with area equal to that of a rectangle? The function radius (equal area) is defined to take as argument an object of class rectangle.

```
#include <iostream>
Using namespace std;
class rectangle/circle
{
Private: int radius;
         int area;
Public;
Class rectangle
  {
   Private: int a,b, Area;
   Public: Int area()
           {
            return a*b;
           }
  };
Class circle
{
```

```
Using namespace std;
Class rectangle
{
 Private: l, b, a;
  void input()
  {cin>>l>>b;
  }
 Public:
    int area()
    {
    return l*b;
    }
};
Class circle
{
Private: float π, r;
float radius of area (rectangle π)
{
  r = sqrt(π.area()/3.14);
  return r;
}
```

```
};
int main()
{
  circle ct
  ct.input();
  circle();
  c.radius of area(ct);
}
```

CONSTRUCTORS AND DESTRUCTORS

- Member functions
- Constructor : To initialize the variables along with object relation.
  → should have the same name of class
  → No return type even void

```
Classname()
{

}
```

- Constructor overloading.
- It is invoked automatically when objects are created
- Allocate dynamic memory to certain objects
- Can have parameters so overloading.
- Implicit executed, it can be called explicitly
- Can have default valued arguments
- Constructs are normally in public but can be declared also in private
- Constructor cannot be inherited.
- Constructors cannot be virtual functions.

eg :
```
class num
{
  int a,b,c;
  public: num()
    {a=3;
     b=2;
     c=5;
    }
};
```

or

```
class num
{ int a,b,c;
  public:
  num();
};
num::num()
 {a=3;
  b=2;
 c=5;
```

Creation of objects.

Class : class name + object name

name obj;

1) Default constructor

Constructor with no parameter.

Problem: Every object has same set of values

2) Parameterised Construction

```
Class num
{ int a,b,c;
  Public: num (int i, int j, int k)
          { a=i;
            b=j;
            c=k;
          }
};
```

Parameters can any type.

(num obj can only be used for default constructors)

num obj (5,10,15); → call by value

⇓

object creation

3) Copy constructor

- Taking value from old objects and giving it to the new object created.

```
Class num
  { int a,b,c;
    Public: num (num & ob)
            {
              a = ob.a;
              b = ob.b;
              c = ob.c;
            }
  };
```

c. objects.

num ob1

num o1 (ob1)

- Possible to pass by reference.

```
class num
  {
  int a,b,c;
   Public: num(num &ob);
  };
```

```
num :: num(num &obj)
{
  a = ob.a;
  b = ob.b;
  c = ob.c;
}
};
```

4] Default Valued Constructor.

```
Class num
{
  int a,b,c;
  Public: num(int i, int j=5, int k=7)
  {
    a = i;
    b = j;
    c = k;
  }
};
```

Creating objects: num o1(10); ✓
num o2(5,7); ✓
num o3(2,7,10); ✓

With one constructor ~~m~~ 3 object relation

- WAP to find the power of a number if no information is passed, it should find 2^3.

```
# include <iostream.h>
using namespace std;
Class power
{
  int a,b;
  Public: Power()
  {
    a = 2*2*2;
  }
  Power num(int a, int b)
  {
    ~~Cout << "Enter 2 numbers";~~
    c = a^b; b;
    cout << c;
  }
};
int main()
{
  int a, b, ob;
```

```
cout<<"Enter two numbers";
cin>>a>>b;
ob. powernum();
ob. powernum(a,b);
cout<<ob.powernum();
}
```

Q2. Write a class when overloaded constructor to find area of a triangle.

```
#include<iostream>
using namespace std;
Class triangle
{
Public: Area(int i, int j)
{
a=0.5*i*j;
cout<<a;
}
Area(float i, float j)
{
a=0.5*i*j;
cout<<a;
}
};
int main()
{ Area ob;
int i,j,a;
cout<<"Enter base and altitude";
cin>>i>>j;
ob.Area(i,j);
cout<<ob.Area;
}
```

Q3 Write a class point which stores two values. Write another class line with that has objects of class point to store start and end points of a line. It should have a member function to return length of the line

```
#include <iostream>
using namespace std;
Class point
  {
   Private:
   int x;
   int y;
  };          Public: read()
Class line              {
  {                      cout<<"Enter two points";
    int length();          cin>>x>>y;
  };
Class line
  {
   point P, Q;
  Public : int length [point P, Point Q];
          { int length
            length = sqrt((pow((Q.x - P.x), 2)) +
                          (pow(Q.y - P.y), 2)));
           return length;
          }
```

```
int main()
{
 int s;
 line l1, l2;
 P. read();
 Q . read();
 S = L.length (P, Q);
 cout << "length is " << S;
}
```

FRIEND FUNCTION:

Some permitted outside functions accessing private data.

Friend + prototype of function.

Access: ob.x ((x private))

eg: Class account

```
{ private : Char name[20];
            int accno;
            float bal;
```

```
Public : void read()
         {
          cin >> name >> accno >> bal;
         }
        friend void showbal (account a);
};
void showbal (account a);
    {
     cout << a.name << a.accno << a.bal;
    }
int main()
 {      char name[10];
  int a, int b;
  cout << "enter name, accountno";
  account a;
  a.read();
  showbal(a);
  return 0;
};
```

- Friend functions cannot access private members directly by name. but by ~~obj~~ using object.
- There should be a declaration

  friend datatype function (obj)
- Usually it has parameter object.

  Declaration can be on private or public section, no difference in meaning
- Friend is not a mutual declaration.

FRIEND FUNCTION

- friend function acting as a bridge
  It must be friend of both classes

eg void f. (X obj1, Y obj2)
&obj1 &obj2


```
{ int temp
  temp = obj1.a
  obj1.a = obj2.b
  obj2.b = temp
}
```

- Friend functions can also take reference parameters parameters
- Friend classes
- X friend of Y : declaration - Y, friend class: X

eg: class beta;
class alpha {
{ Private : int data;
Public : alpha () {data = 90;}
friend class beta;
};
Class beta
{
Public : void f1 (alpha a)
{ cout << a.data; }
void f2 (alpha a)
{ cout << a.data; }
void f3 (alpha a)
{ cout << a.data; }
};

int main ()
{
alpha a; beta b;
b.f1(a);
b.f2(a);
b.f3(a);
}

Q. Only f_1 and f_2 are to be a friend

Class beta;
{ class alpha
{ ~~public~~
Private: int data;
Public: alpha () {data = 90;}
friend ~~friend class void~~ beta
~~friend void beta :: f1(alpha a)~~
~~friend~~
friend void beta :: f1(alpha a);
friend void beta :: f2(alpha a);
};

Q Write two classes and a friend function acting as a bridge between classes to find shaded area

```
int circle
class rectangle
{
  Private: int l, b, a;
  Public: rectarea()
          {
            cout<<"Enter sides";
            cin >> l >> b;
            a = l*b;
          } friend void SArea(rect, circ)
};
class circle
{
  Private: int r; A;
           float A;
  Public: circarea()
          {
            cout << "Enter radius";
            cin >> r;
            A = π * r * r;
          }
};
friend void SArea(rect, circ);
};
void SArea(rect a, circ c)
{
  float SArea
  SArea = a-A;
  cout<<"shaded area"<< SArea;
}
int main()
{
  rect a;
  circ A;
  Shaded(a, A);
  return 0;
}
```

DESTRUCTOR

- Member functions
- Invoked when object is destroyed (Outside the braces - local objects, not static objects
  Scope ends - local and non static object.
- for static objects its life end when program terminates.
  return; exit(); forcefully terminating program.
  Destructor name: ~classname();
- It wont take any parameters
- No return type; not even void type.
- Since no arguments are there, no overloading
- Only one destructor is possible in a class
- It cannot be inherited (for constructor also)
- It can be virtual functions (constructor - cannot be)

```
int count = 0;
class test
{ public: test()
          { count ++;
            cout << "constructor created
            object object no. " << count;
          }
         ~test()
          {
            cout << "Destructor for object"
             << count << " is executed";
            count --;
          }
};
int main()
{ cout << "Inside main";
  test T1;
```

```
{
  cout << " In block 1 ";
  test T2, T3;
}
cout << " Back in main ";
return 0;
}
```

~~→ Constructor created object no 1~~

Inside main

constructor created object no 1

In Block 1

Constructor created for object no. 2

Constructed created object no. 3

Destructor for object 3 is executed

Destructor for object 2 is executed

Back in main

Destructor for object 1 is executed

Operator Overloading

i + j

- operator function written inside class
- return type : use operator(+)(arg)

```
{
}
```

$c_3 = c_1 + c_2$ left side invoked, right : passed

- Class num

```
{
  int x,y;
  Public: num() { }
          num(int j, int k)
          {
            x = j;
            y = k;
          }
  Void show()
  {
    cout << " no = " << x << "y=" << y;
  }
```

```
};
num operator + (num D)
  {
    num temp;
    temp.x = x + D.x;
    temp.y = y + D.y;
    return temp;
  }
};
int main()
  {
    num A(2,3), B(5,7), C;
    A.show();
    B.show();
    C = A+B;
    C.show();
  }
```

num operator + (num D)
{
return(num(x+D.x, y+D.y));
}

works only if 2 arguments.

x = 2 y = 3
x = 5 y = 7
x = 7 y = 10

Only existing operator can be overloaded
arithmetic / relational etc.
ternary operator like ?:, sizeof(), ::,
. etc cannot be overloaded

Binary Operator
- member function - 1 arg
- friend function - 2 arg

Unary operator
- member fn. - No arg
- friend fn - 1 arg

Unary ++

```
Class counter
{
Private : int count;
Public : counter()
        { count = 0; }
```

```
int getcount()
  {
  return count;}
void operator ++()
  { count++;
  }
};
int main()
{ counter c1, c2;
  cout<<"c1="<<c1.getcount();
  cout<<"c2="<<c2.getcount();
  c1++, (since return type is void)
   ++c1;
  c2++;
  cout<<"c1="<<c1.getcount();
  cout<<"c2="<<c2.getcount();
}
```

$c3 = c1++$ $c3 = ++c1$

```
class counter
{ private: int count;
  Public: Counter()
          {count=0;}
```

```
int getcount()
   {return count;}

Counter operator ++()
  { counter s;
    Count ++;     s.count = count;
    return s;
  }
};
int main()
{counter c1,c2,c3;
```

If construction works then only that operation can be done

```
Counter operator ++()
{ count++;
  return (Counter(count));
}
Counter(int c)
{ count = c;
}
```

Counter operator ++() Pre increment case.

```
{ Counter ++;
  return (counter(count));
}
```

dummy argument for distinguishing

```
counter operator ++(int) postfix
{
 counter s;
 s.count = count;
 count++;
 return s;
}
```

OPERATOR VERSION:

```
{ int ic = count;
  count++;
  return (counter(ic));
}
```

⇒ Overload the operator unary − to multiply each element of an object which contains a single dimensional array by −1.

```
Class minus
{
 Private: int a[10];
 Public: read()
        { int i;
          for(i=0;i<10;i++)
            cin>>a[i];
        }
```

```
minus operator -()
{ minus b;
  for(i=0; i<10; i++)
  { a[i] *= -1;
    b.a[i] = a[i]
  }
  return b;
}
int main()
{ int i
  minus a1, a2;
  for(i=0; i<10; i++)
  cout << "a" << a1.read();
  cout << "\n" << a1.minus;
  a2 = -a1;
  cout << a2; }
```

Binary operator - +

```
class pair
  {
   int x,y;
Public:pair () { x=y=0;}
       pair(int p)
       { x=y=p;}
       pair(int p, int q)
       {  x=p; y=q;}

     void display( )
       { cout<<x<<y;
       }

Pair operator +(pair q)
  { pair r;
  r.x = x+q.x
  r.y = y+q.y;
  return r;
  };
```

constructors

```
int main ()
{
  pair A,B,C;
  C = A+B;
  C = A+10;
  C = 10+A;
```

left side object is invoking here no object on left side so, constructor can't be invoked)

Friend function.

Declared defined outside class:

```
Friend pair operator + (pair, pair);
};
pair operator + (pair p, pair q)
 { pair r;
   r.x = p.x + q.x;
   r.y = p.y + q.y
  return r;
 }.
```

In this case

c = 10 + n is also valid.

↓ ↓

r q

Q. Overload the operator + to add the duration in number of days to a date class

```
#include <iostream>
using namespace std;
Class date
 {
  int d;
  int m, y;
  Public: read()
        {
        cout<<"enter a valid date:\n";
        cin >> d >>m >>y;
         }
        display()
        {
        cout<<d<<"-"<<m<<"-"<<y;
        }
        date operator+(int n)
        {
        date ds;
        if (n>365)
        { y+=1
         m=1
        }
         if (n>30)
        {
         if (m==1 || m==3||m==5|| m==7||m==8||m=10
            || m==12)
          {
          n-=31;
           m++;
          }
         else if (m==2)
        {
          n-=28;
          m++;
         }
        else
        {
          n-=30;
          m++;
         }
         if (m>12)
        { y++;
          m=01;
        }
```

```
d = d + n ;
if (d>30)
{
 if (m==4||m==6||m==9||m==11)
  {
   m++;
   d-=30;
  }
 else if (m==2)
  {
   m++;
   d-=28;
  }
 else if (d>31)
  {
   m++;
   d-=31;
  }
 if m>12)
 {y++;
  m=1;
 }
d2.m = m
d2.d = d
d2.y = y
```

```
cout << "New date is: ";
cout << d<<"-" << m <<"-" << y <<"\n";
return d;
}
int main()
{
date d1, d3;
d1.read();
d1.display();
cout << "Enter no. of days to be added"
cin >> n;
d3 = d1 + n;
}
```

Data Conversion (Type Conversion)

1. Basic data type to a class type object
2. Class type object is converted to basic type
3. One class type to another class type object

1. Basic to Class - destination
   source - Constructor - should be 1 arg constructor
   - basic data type - arg

- Constructor in written destination side class.

```
Class time
{
  int hrs;
  int mts;
  Public:
        time (int t)
        { hrs = t/60;
          mts = t%60;
        }
};
Int main()
{
 time T1,
 int dur = 138;
 T1 = dur;
}
```

2) Class to Basic
Some Basic
Done using an operator function

Operator fn. which overloads type cast operator.

Typecast: int i = (int) f;
- int (f)

It should be written as a member function
It should not have arguments
It should not have a return type
→ of dest

```
Operator typename()
{ Operator ins
}
```

vector → scalar.
vector — single dimensional array
scala - ∑ sum of squares

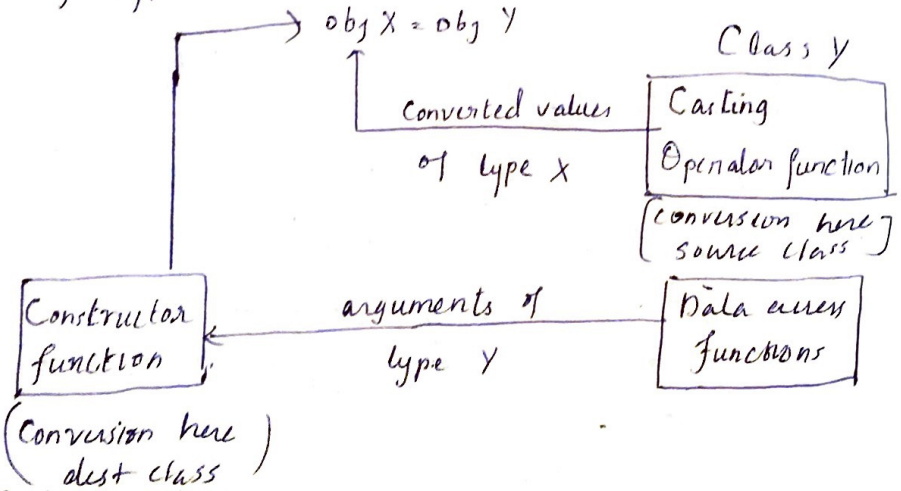
```
Vector class:
Class vector
{ int v[5], size;
  Public: operator float ()
          { for(i=0
            float sum = 0.0
            for(int i = 0; i<5; i++)
```

```
sum+= v[i]*v[i]
return (sqrt (sum))
}
};
Vector v1;
float f;
f = v1;
```

3. Class to Class obj x = obj y

1) Constructor method
written in destination class - 1 arg - Arg type that of Class Y

a) OPERATOR METHOD
written in source class Y, overloads typecast of type destination class X



Invent 1
- Code no.
- Total items in the stock
- Price of each item

Invent 2
- item code
- Value of the item in stock

```
Class invent2;
Class Invent1
{
  int code;
  int items;
  float price;
  Public: Invent1 (int a, int b, float c) //const-
        { code = a;
          items = b;
          price = c;
        }
  void putdata()
  { cout << code << items << price;
  }
  int getcode() { return code; }
  int getitems() { return items; }
  float getprice() { return price; }
/* operator invent2()
  { invent2 temp;
```

```
temp.code = code ;
temp.value = price * items ;
return temp;
} */
};
```

· any method = on conv... / class / last f...

```
operator float ()
{ return items * price ;
}
```

```
Class invent 2
{
 Public : int code;
     float value ;
     invent 2() { code =0; value =0;}
     invent2( int x, float y)
        {code = x; value = y;}
     void putdata()
     {
        cout<<code <<value;
     }
     invent2 (invent1 p)
     {
        code = p.getcode ();
        values = p.getitems () * p.getprice();
     }
};
```

```
int main()
{
  invent1 S1(100, 5, 200.25) ;
  invent 2 D1 ;
  float tot value ;
  /* invent1 → float */
    tot. value = S1;
  /* invent 1 → invent 2 */
    D1 = S1;
  S1. put data();
  cout <<" value "<< tot.value ;
  D1. put data() ;
}
```

CLASS ASSIGN

Consider I/O operations – formatted/unformatted

Output manipulators.