26-7-10

Introduction:-

Computer programming is the art of making a computer do what you want it to do.

C++

Programming language.

language

Alphabet set + Grammar

Programming language

Alphabet set + syntax.

Object oriented -> programming method.

Programming method are classified into

1- Unstructured Programming

2- Structured Programming

structure used for writing a program.

a) Selection

b) Sequential

c) Branching

Lecturenotes.in

if
if else
if else if
nested if] Selection construct/ structure.

Iteration Construct or

Iteration structure or

loop structure.

Lecturenotes.in

Unstructured Programming Paradigm :

Here the main program stands for a sequence of commands or statements which modify data which is global throughout the whole program.

Collection of sequential statements

```
main
{
     st 1;
     st2;
     st3;
```

```
          :
          :
          :
        st n;
    }
```

Drawbacks.

a) Can be used only in very small programs.

b) If the same statement sequence is needed at different locations within the program, the sequence must be copied.

c) If an error needs to be modified, every copy needs to be modified.

Structured Programming:-

a) Procedural Programming

With this, one is able to combine sequence of calling statements into a single phase.

LectureNotes.in

Object Oriented Programming

It's a kind of thinking methodology.

Object

- properties → characteristics
- behaviour. possessed

↓

response shown to differnt actions in environment.

29.7.2010

Charateristics of OOP :-

Object is a tangible entity possessing characteristics & properties.

a) Abstraction

b) Encapsulation

c) Inheritance

d) Dynamic Binding

e) Polymorphism

Lecturenotes.in

a) **Abstraction:-**

It refers to the act of representing the essential features without including background details ~~so~~ or explanations.

b) **Encapsulation:-**

The wrapping of data and related functions into a single unit is known as encapsulation.

c) **Inheritance:-**

Hierarchy:- It is a ranking or ordering of class

- is a - relationship - its class structure - inheritance
- part of - relationship - its object structure - aggregation.

It is the process by which objects of one class acquires properties of objects of another class.

LectureNotes.in

d) Dynamic Binding:-

- Also called as late binding. The code associated with a given message will be decided at run time.
- Helps achieving polymorphism.

e) Polymorphism:-

→ It is the property that allows to exhibits different behaviours at different situations.

→ Different objects respond differently to a same message.

ex:-

Message draw

- circle responds with drawing of a circle.
- polygon responds with drawing of a polygon

Lecturenotes.in

Object Oriented Programming :-

It is the method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and those classes are all members of hierarchy of classes united via inheritance relationships.

- Objects are the basic building blocks not algorithms.
- Each object is an instance of some class.
- objects interact with each other
- classes are related via inheritance.

Lecturenotes.in

30.7.10

→ Write a C program for displaying Hello students.

```
  #include<stdio.h>
int main( )
  {     //output section
        printf("Hello students");
   }    printf("End of main function");
        return 0;
```

//name of program

Preprocessor is a program that manipulates a program before compilation. Normally it includes header files

```
#include<iostream>
            └→It contains all input,
              output related objects.
int main()
{
      cout<<" Hello students" <<endl;
```

cout is an object of iostream class, which is used to display output in monitor.

Lecturenotes.in

<< - insertion operator; is a binary operator using 2 operands - cout and string or constant.

```
    cout<<"\n End of main";
    return 0;
}
```

filename - $ vi filename.cpp

<u>For compilation:-</u>

```
$ c++ filename.cpp.
```

<u>For execution:-</u>

```
$ ./a.out
```

For newline we use either \n or endl.

endl → manipulator

\n → escape sequence

```
int main
{
        int data=10,data1=20;
        cout<<data<<data1;
        cout<<"end of main";
}       return 0;
```

LectureNotes.in

```
// To print name, rollno, sicno, sgpa //
#include <iostream>

int main()
{

      int roll, sic; ~~sgpa;~~ float sgpa
         char name; // declaration.
                    // of variables
      char name = "Jyotsna_Dash";
            roll = 20;
            sic = 22092710 ;
            sgpa = 8.75 ;

       cout<<"\n My name is: "<<name;
       cout<<"\n Rollno: "<<roll;
       cout <<"\n SICNO: "<<sic;
       cout<<"\n 1st sem sgpa: "<<
                              sgpa;
       cout<<"\n end of main() ";
       return 0;
} // end of main()
// end of program
```

Lecturenotes.in

2.8.10

Taking Input from keyboard:-

An object cin of iostream class with extraction operator(>>) is used.

Syntax:-

```
cin>>variable;
```

```
cout<< variable1<<variable2;
```

↳ cascading of insertion operator.

```
cout<<23;
cout << a;
cout<<" silicon";
cout <<"\n";
cout << endl;
```

↳ manipulator

eg

```
int var1;
cout<<"\n Input variable value:";
cin>> var1;
```

Lecturenotes.in

```
char ch[10];
  cin>>ch; // only a single
                 word input.

 int ch[10];
    int i;
 for(i=0; i<10; i++)
      cin>>ch[i];
```

Q/ Write a C++ program to take the contents of an integer array and display the contents */

Sol:-

```
 // start of program
  #include<iostream>
 // start of main()
 using namespace std;
 int main()
 {
      // Declaration of variables
         int arr[10], i;
      // Taking input to array
         cout<<"\n Enter elements:";
```

Lecturenotes.in

```
for(i=0; i<10; i++)
      cin>> arr[i];
//Display of array contents
 cout<<"\n Array elements:";
 for(i=0; i<10; i++)
      cout<<"\t"<<ch[i];
return 0;
} //end of main()
 //end of program
```

Namespace:-

Is a logical concept to localize the names/identifiers.
std is a default namespace.

5.8.10

Datatypes:-

1) **Bool**

Size- 1 byte

Range- (0,1).

LectureNotes.in

2) w. char_t

used for white character

Size- 2 bytes

Range-$(0-(2^{16}-1))$

Variables:-

Same rules as in C.

Operators:-

new, delete, :: (scope resolution operator)

Scope Resolution Operator:-

→ To access global variables

→ Defining scope of a variable.

→
```
int i=10;
main()
{
    int i= 20;
    {
        int i=30;
        printf("%d", i);
    }
}
```

Lecturenotes.in

O/P → 30.

→ If it is written

```
int i=10;
main()
{
   int i=20;
      {
block      int i=30;
           ~~RO~~ cout<< i<<::i; —— ①
      } cout<< i<<::i —— ②
   return 0;
}
```

::i → always access global variable above main.

O/P :- ~~2~~ 30 $\overset{10}{\underline{20}}$. (for statement 1)

O/P :- 20 10 (for statement 2).

Lecturenotes.in

new → creates memory dynamically and delete deletes memory allocated dynamically.

Reference Variable:-

It is an alternative name for another variable. Also called alias.

Syntax:-

```
datatype &Reference vname =
                     vname;
```

eg:-

```
int var = 10;
   (var = 20;)
(int var1;)
   int &var1 = var;
```

Function call by Reference:-

Rtype fname(datatype);
call by value

Rtype fname(datatype);
call by ~~ref~~address

Rtype fname(data type&);
call by refuence

Q Swap two variables(float)

```
void swap(float&, float &);
int main()
{
    float var=2.5, var1=5.4;
    swap(var, var1);
```

Lecturenotes.in

```
cout<<"\n swapped values:"<<
               var<<"\t"<< var 1;
 return 0;
}
void swap(float &v1, float &v2)
{
       float temp;
        temp = v1;
         v1 = v2;
         v2 = temp;
  }
```

Output:- 5.4 2.5

New & delete :-

→ New is an operator that is used to create memory dynamically.

Syntax:-

```
       datatype *ptr;
            ptr = new datatype;
```

eg:-

```
            int *ptr;
            ptr = new int;
             char *ptr;
              ptr = new char;
```

Lecturenotes.in

Q Create array dynamically.

```
int *ptr;
ptr= new int[10];
```

↓

allocation of memory to an array of 10 integer elements.

General syntax:-

```
datatype *ptr;
ptr= new datatype[size];
```

6.8.10

Problem:-

1)
```
int main()
{
  int var= 20;
    int &var1=20; ←  reference to constant is not allowed.
      Cout<<var1;
  return 0;
}
```

O/P → garbage value./error

Lecturenotes.in

2) int main()

```
{
      int var=20;
      int &var1=var;
      int &var2=var1;
      ~~int~~ cout<< var2;
      return0;
}
      O/P → error.
```

Output:- ← cannot create a reference to a reference variable.

3) int main()

```
{
      int var, *ptr;
      ptr= &var;
      *ptr=20;
      int &var1=ptr;
        cout<< ~~*ptr~~ *var1;
      return0;
}

O/P → error.
```

cannot create a reference to a pointer not allowed

Lecturenotes.in

Q Swap two ~~integer~~ float values (created dynamically).

Progam:-

```
#include<iostream>

 using namespace std;
 void swap( float *, float *);
 int main()
  {

      //declaration of pointers.
       float *ptr, *ptr1;

      // Dynamic memory allocation

       ptr= new float(2.6);
                       ↑      ↳ value.
                   initialisation
       ptr1 = new float (22.6);

      ~~swap(~~
      // function call
       swap(ptr, ptr1);
      Cout<<"\n the swapped
                    values:";
      cout<< *ptr <<" \t"<< *ptr1;
     return 0;
  } // end of main()
```

Lecture Notes .in
Lecturenotes.in

```
// function defination
void swap (float * ptr, float * ptr1).
{
        float temp;
        temp = * ptr;
          * ptr = * ptr1;
          * ptr1 = temp;
   }
   // end of the function
// end of program
```

Output:-

```
The swapped values: 22.6 2.6
```

Q. Reverse the element of an array where the array is created dynamically.

Sol:-

```
// start of program
# include< iostream>
using namespace std;
# define size 10
```

```
int main()
{
   //declaration section
    int *ptr; int i=0,j=size-1;
   // dynamic memory allocation
    ptr= new int[size];
   // loop for ~~reverse~~ input
    for( ; i<size;i++)
    {
       cin>>*(ptr+i);
```

we use '( )' to maintain priority

we cannot write cin>> ptr+i because we can't ~~use~~ provide the address of a variable to cin coz it takes value of variable.

```
}
```

Lecturenotes.in

```
// Revereing the array

~~for~~ i=0, j=size-1;
   L ←———— R
            Associativity of
            comma operator

~~for~~ while(i<=j)
   {
        int temp;
        temp= *(ptr+i);
        *(ptr+i) = *(ptr+j);
        *(ptr+j)= temp;
   ~~}~~   i++;
        j--;
   }

   ~~for~~
   // output section
   for(i=0; i<size; i++)
        cout<< *(ptr+i);
   return 0;
}
// end of main()
// end of programe.
```

Lecture Notes .in
LectureNotes.in

Assignment :-

Q- Create an array dynamically and find duplicate eliments of the array.

Creating 2-D Array dynamically :-

```
// start of program
#include <iostream>
using namespace std;
#define size 10
int main()
{
     // declaration section
        int (*ptr)[size]; ← pointer
                            to 2-D
                            array
     // Dynamic memory alloc^n
        ptr = new int[size][size];
     // Input section
     int i=0, j=0;
     for( ; i<size; i++)
     {
          for( ; j<size; j++)
               cin>> ptr[i][j];
     }
```

Lecturenotes.in

```
// Display section
 for (i=0; i<size; i++)
 {  for(j=0;j<size;j++)
          cout<< ptr[i][j];
 }
 retern 0;
} // end of main()
// end of program.
```

Sol:-

```
  // start of preogram
  #include<iostream>
  using namespace std;
  # define size 10
  // start of main()
  int main()
  {
      // Declaration Section
      int *ptr, i=0, j=0, ctr=0;
      // Dynamic Memory Alloc^n
      ptr= new int[size];
```

LectureNotes.in

```
// Input section
for(   ; i<size;i++)
{

        cin >> *(ptr+i);
}
// finding duplicate
  if( ~~*(ptr~~.ptr[i]==ptr[j])
  {
        ctr++;
        i++;
        j++;
     }
// Output section
   cout<<"\n Duplicate elements"
                      << ctr;
   return 0;
}
 // end of main()
  // end of program
```

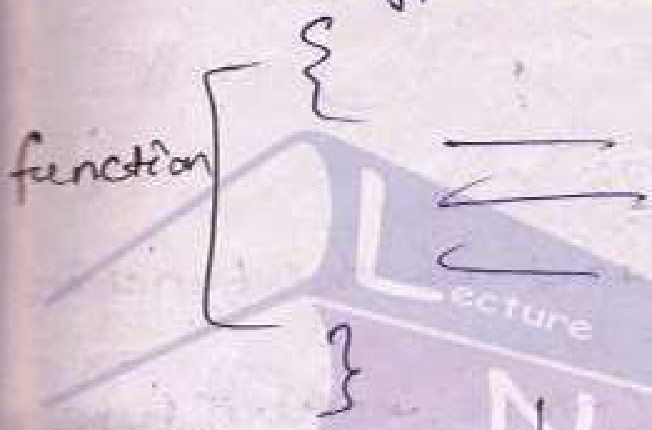
Lecturenotes.in

9.8.10

Functions, Default Arguements, Overloading:-

Function → self-contained block to do a specific task.

rtype name (arguements)



↓

modular programming.

Lecturenotes.in

Function

↓ Prototype ↓ call ↓ Defination

Function prototype:-

a) no. of arguements
↳ variables through which the data is recieved.

b) type of arguments

c) function name

d) Return type.

Function call:-

Function is called to perform the task.

Function defination:-

It defines the task of the function

Function header(name, return type & formal arguments) followed with a block

Q Write a function to add two integers.

Sol:-

```
//start of program
#include<iostream>
using namespace std;
// function prototype
int add(int, int);
```

Lecturenotes.in

```
// start of main()
int main()
{
   // Declaration section
     int var, var1, var2;
   // Input section
     cout<<"\n enter the 2 nos:"
       cin>>var>>var1;
   // Function call            actual
                               parame
   var2= add( var, var1);
   // Output section
     cout<<"\n Addition:"<<var
     return 0;
} //end of main()

// Function defination
   int add(int x, int y)
   {                          formal
        return (x+y);         paramete
   }

// end of program
```

Lecturenotes.in

when the function is called. then it is associated with the address and is called as <u>function binding</u>

If the binding is done by the compiler at compilation time, then it is called as <u>early binding</u> or <u>static binding</u>.

<u>Default argument:-</u>

It assigns a parameter ~~as~~ a default value when no argument corresponding to that parameter is specified in a call to that fun^n.

Default value assignment is always from <u>R-to L</u> i.e. the rightmost formal argument will be assigned with the default value then the next value.

Lecturenotes.in

Function Overloading:-

- It is the process of using the same name for two or more functions.
- Each redefination of the function must use either <u>different types of arguments</u> or <u>different no. of arguments.</u>
- It is only through these differences that the compiler knows which fun^n to call in any given situation

eg:-

```
int add(int, int);
float add(float, float);
                    function overloading.

add(a,b); // a, b are integers
add (x,y); // x,y are float
```

Compiler checks the <u>type of argument</u> and binds with the appropriate function.

Lecturenotes.in

```
int add( int , int);
   void add(int);

   add( a, b);
     add( x);
```

Here compiler distinguishes by the no. of arguments

12.8.10

Q Overload function abs() to find absolute value of

1) integer
2) float.

Sol:-

~~// Prototype declaration~~
~~int abs(int);~~
~~float abs(float);~~

```
// start of program
 #include <iostream>
  using namespace std;
 // Prototype declaration
  int abs(int);
  float abs(float);
```

Lecturenotes.in

```
// start of main
 int main()
 {
        //Declaration section
         int var;
         float var1;
        // Input section
         cin>>var;
         cin>>var1;
       // output section
        cout<<"\n Absolute value
                    are:";
        // calling functions
        cout<<abs(var);
        cout<<abs(var1);
        return(0);
 }
// end of main()
 // function defination
 int abc(int x)
 {
     return x<0? -x:x;

 }
```

Lecturenotes.in

```
float abs ( float x)
{
    return x<0.0?-x:x;
} // end of defination.
// end of program
```

Q- overload functions stradd() function to concantenate

1) 2 strings
2) Append str1 with str2 from a given ~~8th~~ character/index (index =8).

sol:-

```
// start of program
#include<iostream>
using namespace std;
// prototype fun^n declaration
void stradd(char*, char*);
void stradd(char*, char*,
                    int);
```

Lecturenotes.in

```
// start of main()
 int main()
 {
      // Declaration section
        char *ptr, *ptr1;
         int  i, index;
     // Dynamic memory allocation
        ptr = new char[80];
        ptr1 = new char[80];
      // Input section
        ~~cin>> ptr;~~ cout<<"\n Input 1st
        ~~cin>> ptr1;~~          string:";
          cin>>ptr;
        cout<<"\n Input 2nd string:";
          cin>>ptr1;
    // Function call of 1st stradd
        stradd(ptr, ptr1);
         cout<<ptr; cout<<ptr1;
    // Procedure for 2nd part
          cin>>ptr; cin>>ptr1;
         i=0, index=0
        while(*(ptr+i)!=' ')
       {
               index ++;
       }
               i++;
```

Lecturenotes.in

```
//function call of 2nd stradd
   Stradd( ptr, ptr1, index+1);

   // Display section
       cout<< ptr;
       cout<< ptr1;

   return 0;
} // end of main()

// Defination of 1st function.
void stradd(char *ptr, char *ptr1).
{
     //Declaration section
     int  i, j, len1, len2;
     // calculation of length of 1st
             string.
           i=0; len1=0;
       while(*(ptr+i) != '\0')
       {
             len1++;
             i++;
        }
```

Lecturenotes.in

```
// calculation of length of 2nd
      string
        j=0, len2=0;
   while( *(ptr+j)! = '\0')
   {
          len2++;
           j++;
    }

// concatenation of 2 strings.
         i= 0; ptr[len1]= ' '; len1++;
    while( *ptr1[i] ! = '\0')
    {
          *(ptr+ len1+ i) = ptr1[i];
           i++;
     }
       ~~ptr[i] = '\0';~~
       ptr[len1+i]= '\0';
} // end of 1st function defination
```

Lecturenotes.in

```
// Defination of 2nd function

void stradd(char *ptr, char *ptr1,
                       int index)
{

    // Declaration section
     int i, j, len1, ~~len2~~;
     ~~(len1= strlen(ptr);)~~
     len1= strlen (ptr1); ~~len1 —~~,
     ~~i= index;~~
    ~~// Concantenation~~
     ~~while(ptr[i]!='\0')~~
     ~~{~~

// assign blank space at the end
       of 2nd string

     ptr1[len1]=' ';
       len1++;
      i= 00; index;
// concantenation
    while( ptr[i]!= '\0')
    {
        ptr1[len1+i]= ptr[i];
          i++;
    }
    ptr1[len1+i]= '\0';
} // end of 2nd function definati
// end of program
```

Lecture Notes .in
Lecturenotes.in

19.8.10

How to define a class?

Syntax:-

```
class classname
{
        members;

};
```

members → Data members (attribute)

member fun's (behaviour)

→ public, private → Access specifiers

Q- Define a class student with data members- name, rollno & cgpa & member fun's- input and output.

Sol:-

```
//Defination of class
class student
{
    private:
        char name[20];
```

Lecturenotes.in

```
    int rollno;
    float sgpa;
public:
    // input section
    void input()
    {
        cout<<"\n enter name:";
        cin>>name;
        cout<<"\n input rollno:";
        cin>>rollno;
        cout<<"\n input sgpa:";
        cin>>sgpa;
    }
    // end of input function
    void output()
    {
        cout<<"\n Members of class:"
        cout<<"\n Name:"<<name;
        cout<<"\n Rollno:"<<rollno;
        cout<<"\n sgpa:"<<sgpa;
    }
    // end of output function
}; // end of class
```

Lecturenotes.in

Access specifiers define the mode of accessing any members of a class.

Private members are accessed only by the member functions of the same class.

Public members can be accessed by any function of the program

By default, if no access specifiers are specified, then the access becomes private.

Grouping the members of a class into a single unit is called as encapsulation.

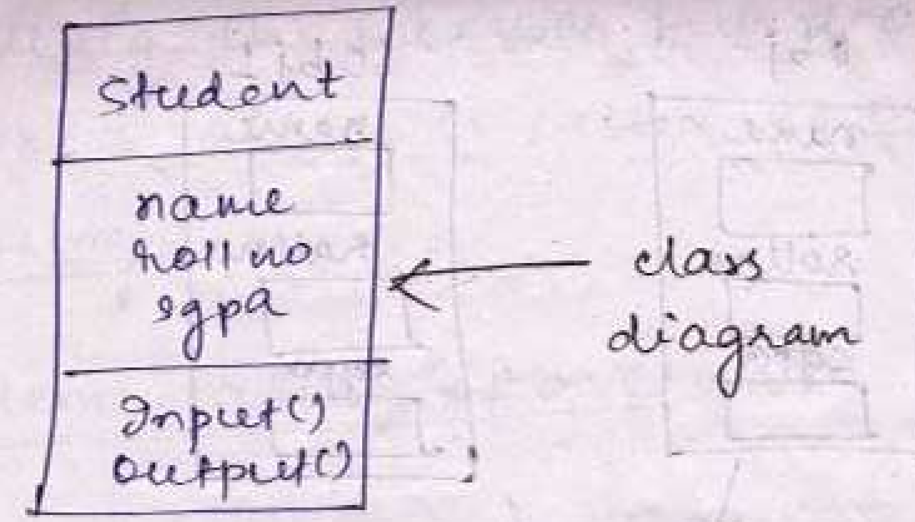
class defination → logical concept

object of a class created

↓

memory allocated

Lecturenotes.in



Defining object of a class:-

Syntax:-

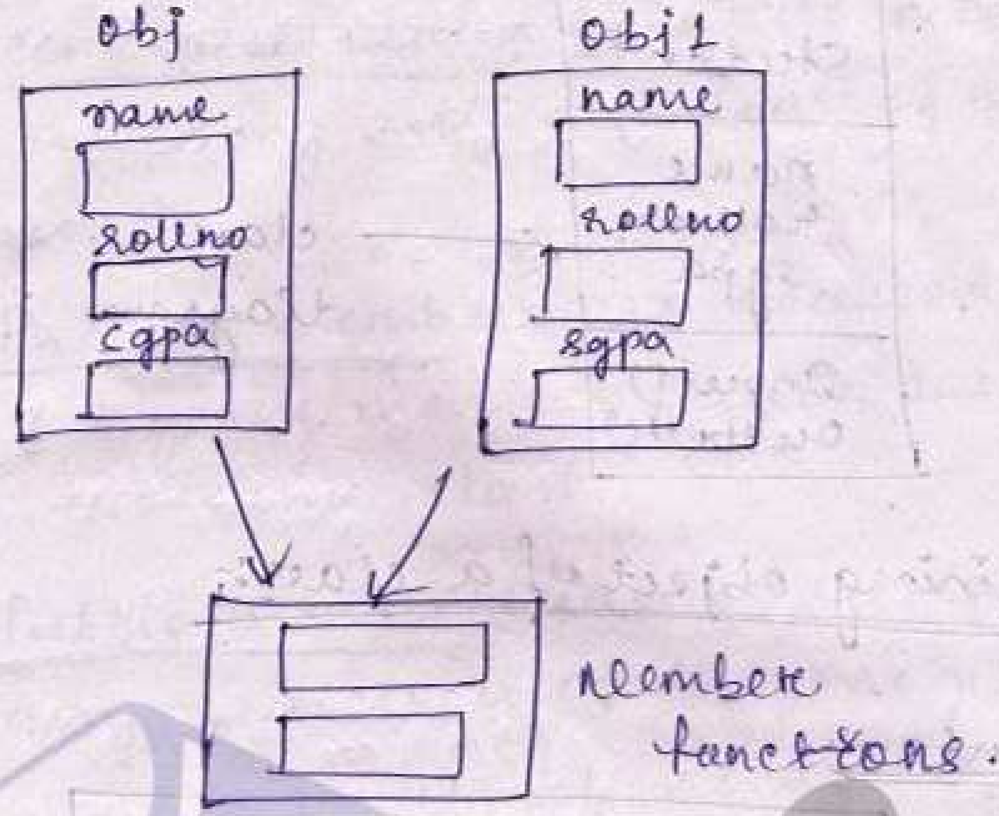
```
classname objectname;
```

eg→ student obj;

// obj is an object of type student or
obj is a variable of type student

student obj1;

LectureNotes.in



Accessing members:-

Syntax:-

```
objectname. membername;
```

eg →

obj. name;
obj1. name;
obj. input;
obj. output;

Message passing

Lecturenotes.in

Calling the member function for a object is called as <u>message passing</u>.

Implementation of a class:-

eg:-

```
// class defination.

#include <iostream>
using namespace std;
int main()
{
     student obj;
    cout<<"\nTake input for
          objects of a class:";
     cin>> obj.name;
    //Incorrect! Name is a
         private member
     obj.input;
```

Lecturenotes.in

```
~~Obj. output()~~;
cout<<"\n Displaying contents of
          the object";
 Obj. output();
return 0;
}
//end of main()
//end of program
```

Defining a member fun^n outside class:-

Syntax:-

Lecturenotes.in

```
~~class~~
returntype classname::member
          functionname(arguments)
{
    //statements
}
```

eg→

```
class student
{
    private:
       char name[10];
    public:
       void input();
};

void student:: input ()
{
    cout<<"\n Enter name:";
    cin>>name;
}
```

specifies/ navigates/ associates a member func^n to the class

20.8.10

eg→

```
#include <iostream>
using namespace std;
class student
{
    private:
      char name[20];
      int rollno;
      float sgpa;
```

Lecturenotes.in

```
  public:
      void input();
      void display();
 };

// Defining member functions
  void student :: input()
  {
      cout<<"\n Enter the data";
        cin>> name;
        cin>> rollno;
        cin>> sgpa;
  }

  void student :: display()
  {
      cout<<"\n Student informations";
       cout<<name<<endl;
       cout<< rollno<< endl;
       cout<< sgpa;
  }

// end of function defination
```

Lecturenotes.in

```
// start of main
int main()
{
    // Creating objects
    student obj1, obj2;
    // Input section for obj1
     obj1. input();
     // Display section for obj1
     obj1. display();
     // Assigning values of obj1 to obj2
      obj1 = obj2;
        └→ But both the objects
           need to be of the
           same class / type.

      obj2. display();
     return 0;
  }
  // end of main()
```

Lecturenotes.in

Passing objects to non-member functions

void fun(student);

Syntax:-

```
returntype function (classname);
           name
```

eg:-

```
// start of program
#include<iostream>
using namespace std;
// Prototype declaration
void fun(student);
// start of main
int main()
{
    // Declaration section
    Student object;
    // Input section
    Object input();
```

Lecturenotes.in

```
    //calling function
    fun( object);
    return 0;
  } // end of main
 // Function defination
 void fun(student obj)
 {
       obj. display();
       cout<<"\n end of function";
 }
 // end of program
```

functions returning objects:-

Syntax:-

```
classname functionname(arguments);
```

↓

The type that will be returned by the function

Lecturenotes.in

```
// Start of program
#include<iostream>
using namespace std;
// Prototype declaration
student fun(student);

// start of main
int main()
{
      // Declaration section
      student object, obj2;
      // Input section
       object input();
      // calling function
       obj2 = fun(object);
       return 0;
} // end of main
// Function defination
Student fun(student obj)
{
          obj. display();
```

Lecturenotes.in

```
        return obj;
  }
  //end of function defination
 // end of program
```

Object Pointer:-

```
// start of program
#include<iostream>
 using namespace std;
// start of main
 int main()
  {
     // Declaration section
       student obj, obj1;
      student *ptr;
       ptr = &obj;
      // Calling input for obj
       ptr -> input();
```

LectureNotes.in

```
// calling dicplay for obj
  ptr -> dicplay();

 // Ptr now pointing to obj1

   ptr = &Obj1;

 // calling input for obj1
    ptr -> input();

 // calling display for obj1
      ptr -> display();

  return 0;

}

// end of main

// end of program
```

LearnNotes.in

This pointer:-

```
void student::input()
 {
       cin>> this -> name;
       cin>> this -> rollno;
```

cin>> this ->sgpa;

25.8.10

It is a pointer which contains the address of currently accessed object.

i.e; when a member function is invoked by any object, then this pointer is passed to the member function implicitly containing the address of the object who calls the member function.

eg:-

```
//start of program
# include<iostream>
using namespace std;
// class defination
class employee
{
     private:
          char name[20];
           int age;
     public:
          void getdata();
          void putdata();
 };
```

Lecturenotes.in

```
// Defining the member function

Void employee::getdata()
{
    cout<<"\n Enter name:";
      cin>>name; // this->name
    cout<<"\n Enter age:";
       cin>>age; // this->age
}

Void employee::putdata()
{
     cout<<"\n Name:'<<name;// this->name
     cout<<"\n Age:"<<age;
                    // this->age
}.

// end of defination

// start of main()
```

Lecture Notes
LectureNotes.in

```
int main()
{   // Declaration section
    employee obj1, obj2;
    ~~cout<<"\n I~~

    // Input section
    cout<<"\n Input data";
     obj1.getdata();
     obj2 :getdata();
    // Display section
      cout<<"\n output:";
       obj1.putdata();
      obj2.putdata();
      return 0;
  } // end of main

 // end of program
```

lecturenotes.in

This pointer is used only for members function.

Array of objects:-

```
// start of program
# include<iostream>
using namespace std;
# define size 5
//class defination
class employee
{
    private:
        char name[80];
        int age;
    public:
        void getdata();
        void putdata();
};
// Defining the member function
void ~~getdal~~ employee:: getdata()
{
    cout<<"\n Enter name:";
    cin>> name;
    cout<<"\n Enter age:";
    cin>> age;
}
```

Lecturenotes.in

```
void employee::putdata()
{
    cout<<"\n Name:"<< name;
    cout<<"\n Age:"<< age;
}
// end of defination
// ~~end~~ start of main
int main()
{
    // Declaration section
    employee objectarray[size];
    // Input section
    cout<<"\nInput data to array:";
    for(int i=0; i<size; i++)
        ~~cin>>~~ objectarray[i].getdata();
    // output section
    cout<<"\n Display";
    for(i=0; i<size; i++)
        objectarray[i].putdata();
    return 0;
} // end of main // end of program
```

Lecturenotes.in

Static Datamember:-

Syntax:-

```
static datatype varname;
```

→ Initialised to zero.

→ Stored in memory

→ Local scope

→ But remains till execution of end of program

- A static data member is declared using the keyword static.
- It has to be defined outside the class:

Syntax:-

```
type classname:: v-name;
```

lecturenotes.in

only one copy of

- A static variable ~~on~~ is created and it is shared by all the objects of the same class (i.e; the objects of the class will not have indivitual copy of static data member).

26.8.10.

eg

```
// start of program
#include<iostream>
using namespace std;
// class defination
class item
{
   private:
      int itemid;
      float itemprice;
      static int count;
   public:
      void setdata(int, float);
      void display();
};
// end of class defination
```

Lecturenotes.in

```
// Member function defination

void item :: setdata(int x, float y)
{
        itemid = x;
        itemprice = y;
}

void item :: display()
{
        cout<<"\n Item id:"<<itemid;
        cout<<"\n Item price:"<<itemprice;
        cout<<"\n"<<count;
}

// end of definations

// Defination of the static data memb
    int item:: count;=10;

// Start of main()
int main()
{
```

Lecturenotes.in

```
    // Declaration section
      item obj;
    // Input section
      obj.setdata(1, 100.25);
    // output section
      obj.display();
    return 0;
  } // end of main()
// end of program
```

Lecturenotes.in

Output:-

```
  Item id: 1
  Item price: 100.25
      0 // 10
```

→ when a class contains a static variable only one copy of the static variable is created and i.e; shared by all the objects of the class. and it is created before the creation of any object of the class.

Static member function :-

→A member function can also be declared as static.

→A static member function can access only static members of the class.

eg

```
//start of program
#include<iostream>
using namespace std;
//class defination
class item
{
    static int count;
     int itemid;
  public:
    ~~static~~ void setdata();
    static void show();
};
```

Lecturenotes.in

```
// Member function defination

void item :: setdata()
{
        count = 10;
        Itemid = 1;
}

void item :: show()
{
        cout<<"\n"<<itemid; — X
        cout<<"\n"<<count;
}

// Defination of static member
   int item :: count;

// start of main
int ~~item~~ main()
{
      // Declaration section
        item obj;
      // Input section
        obj.setdata();
```

Lecturenotes.in

```
// Output section
    obj.show();
    item::show();
        ↳ show can be called
          independent of any
          object using classname
    return 0;
}
// end of main()
// end of program
```

Output:-

10

Lecturenotes.in

Assignment:-

1- Define a class employee with memb

→ employee name

→ basic salary

→ DA

→ Gross (Basic + $\underbrace{(\text{Basic} \times 0.15)}_{\text{HRA}}$ + D

→ structure address
↳ 1) city name
2) plot no
3) pin

and member functions

Input(),
calculate() → gross salary.

and display()

~~Ecto~~
Sol :-

```
// start of program
# include<iostream>
using namespace std;
// structure defination
struct address
{
      char city_name[20];
      ~~flo~~ int plot_no;
        int pin;
};
```

Lecturenotes.in

```
// class defination
class employee
{
        char emp_name[20];
        float basic_salary;
        float DA;
        float Gross;
        struct address add;
     public:
        void Input();
        float calculate();
        void display(float);
};
//Member function defination
void employee:: input()
{
     cout<<"\n Enter employee name:";
      cin>>emp_name;
     cout<<"\n enter basic salary:";
      cin>> basic_salary;
     cout<<"\n enter DA:";
      cin >> DA;
```

Lecturenotes.in

```
    cout<<"\n Enter address:";
    cout<<"\n Enter cityname:";
     cin>>add.city_name;
    cout<<"\n Enter plot number:";
     cin>> ~~plot~~.add.plot_no;
    cout<<"\n enter pincode :";
      cin>> add.pin;
}
float employee::calculate()
{
      ~~float~~
       gross = basic_salary +
                 (basic_salary * 0.15) +
                  DA);
       return(gross);
}
void employee:: display(~~g~~float gs);
 {     ~~float gs;~~
      cout<<"\n Employee details:";
      cout<<"\n Employee name:"
                  << emp_name;
      cout<<"\n Basic salary:"<<
                        basic_salary;
```

LectureNotes.in

```
    cout<<"\n Daily Allowance:"<< DA;
    cout<<"\n Address :";
    cout<<"\n city name:"<< add.
                                cityname;
    cout<<"\n Plot no:"<< add.plotno;
    cout<<"\n Pin code"<< add.pin;
    cout<<"\n Gross salary "<< gs;
                        ~~calculate()~~;
}
// end of defination
// start of main()
 int main()
 {
     // Declaration section
        Employee emp; float gs;
     // Input section
        emp.input();
     // calculation of gs
        gs = emp.calculate();
```

LectureNotes.in

```
  // output section
 emp.display(ge);
     return 0;
  }
  // end of main()
 // end of program.
```

30.8.10

Constructors :-

→ A special member function used to create and initialise the object to a class.

→ The name of the constructor function is same as the class name.

→ It has no return type and it may take arguments.

LectureNotes.in

eg→

```
// start of program
// class defination
class myclass
 {

      private:
          int a,b;
      public:
          void input();
         void display();
  };   myclass ( );← constructor

// Member function definations
void myclass::input()                  Repla
 {
      cout<<"\n Input data";
      cin>>a>>b;

 }

void myclass::display()
 {
      cout<<"\n Members of class:";
      cout<<"a<<"\t"<<b;
 }
```

Lecturenotes.in

```
// start of main()
int main()
{
          // Declaration section
          myclass obj; // If included
                                          constructor
we don't [ // Input section               is called
need     [ obj.input();                 automatically
this
anymore  // Output section               to create
                                          one &
          obj.display();                  initialize
                                          the data
          return 0;                       members.
}
// end of main()
// end of program.

*// Defination of constructor
myclass::myclass()
{
       cout<<"\n Inside the
                       contructor";

          a =10;
          b = 11;
}
```

LectureNotes.in

Default Constructor:-

A constructor which takes no arguments is called as default constructor.

Parameterized Constructor:-

A constructor which takes arguments is called as parameterized constructor.

→A class can have both default as well as parameterized constructor.

eg:- // class defination

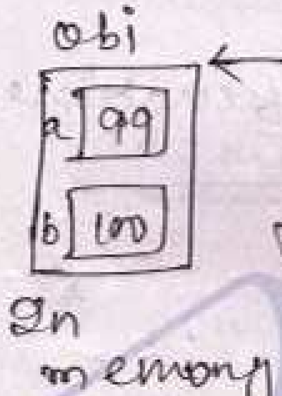
```
class myclass
{
    private:
        int a,b;
```

Lecturenotes.in

```
public:
        myclass();
        myclass(int, int);
        void display();
};
// Defination of member functions

myclass:: myclass()
{
        a=0, b=0;
}

myclass:: myclass(int x, int y)
{
        a=x;
        b=y;
}
void ~~disp~~ myclass::display()
{
     cout<<"\n Values are:"~~<<~~;
     cout<< a<<"\t"<<b;
}
```

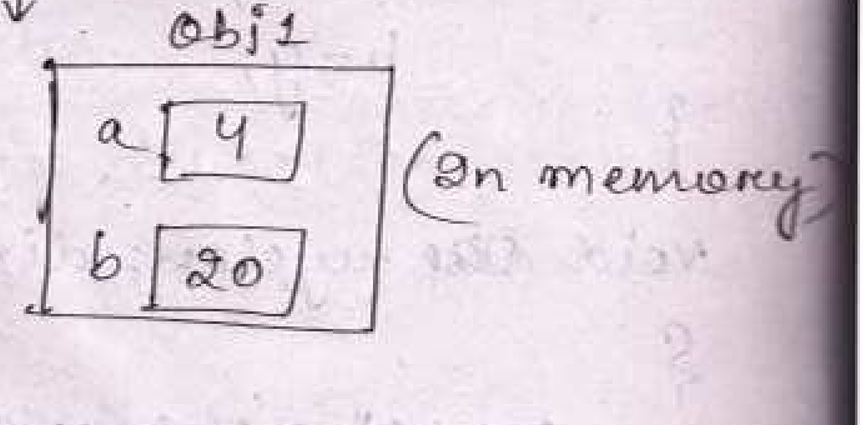
Lecturenotes.in

```
//start of main()
 int main()
 {
      // Declaration function
      myclass obj(99, 100);
```



The parameterized construc-tor will be invoked with the values supplied in the argument list. to creat and initialise the object.

```
     myclass obj1(4, 20);
```

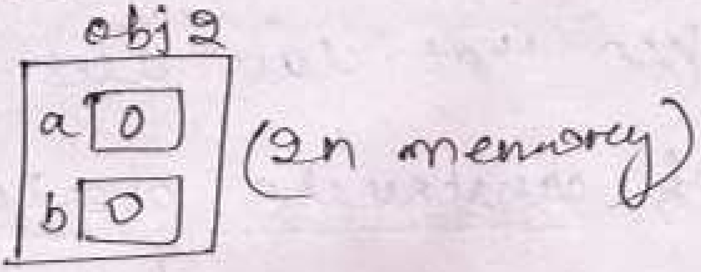


Lecturenotes.in

```
myclass obj2;
```

↓

Default constructor will be invoked.



```
// output section
      obj.display();
      obj1.display();
      obj2.display();
   return 0;
} // end of main()
```

LectureNotes.in

Output:-

```
Values are: 99 100
Values are: 4 20
Values are: 0 0.
```

Constructor Overloading:-

→ when a class uses both default and parameterized constructor then that class uses the concept of constructor overloading.

→ when a class doesnot have any constructor then the compiler will create and provide a default constructor

2.9.10

Lecturenotes.in

Copy Constructor:-

→ It is used to initialise the object of a class with value of another object of same class.

→ Syntax :-

```
classname ( classname &varname)
{
        Body of constructor

   }
```

↳ always takes a reference to the object name

→ The argument to the copy constructor is reference to object of same class.

eg →

```
// start of program
# include<iostream>
using namespace std;
// class defination
class myclass
{
      private:
      -  int a,b;
     public:
        myclass()
       {  a=0;b=0; }
```

LectureNotes.in

```
myclass(int x, int y)
{
      a = x, b = y;

  }

 myclass(myclass &obj)
 {                 'a' of object
                       newly create
      a = obj.a;
      b = obj.b;
 }
                   'a' of object
               passed as a parameter

Void display()
{

      cout<<'a<<"\t"<<b;

}

}; // end of class defination
```

Lecturenotes.in

eg:-

```
// class defination

class student
{

    char *name;
     int roll;
   public:
     student()
     {
         name= NULL;      NULL pointer
                          points to
                          nothing
         roll=0;
     }
     student(char *ptr, int x)
     {
          name= new char[size];
          strcpy(name,ptr);
           roll=x;
      }
     void display()
     {
          cout<<name;
           cout<<roll;
      }
}; //end of class defination
```

Lecturenotes.in

```
//start of main()

int main()
{

  // Declaration section

   student obj("silicon", 20);
```

↓

we can call the parameterize
conctructor in this way or we
can write

```
      char name1[size];
        cin>> name1;
      int y = 90;
      student obj1(name, y);

   // output section

        obj. display();
        obj1. display();
       return 0;
}

// end of main
```

Lecturenotes.in

Creating object dynamically:-

eg→ // class defination

```
class myclass
{
    int a,b;
  public:
     myclass ()→ Default
     {                parameter
        a= 0;
        b=0;
     }
     myclass(int x, int y)
     {                 └→ parameteri-
        a=x;                 zed
        b=y;            constructor
     }
     void display()
     {
        cout<< a<<b;
     }
}; // end of class defination
```

Lecturenotes.in

```
//start of main
int main()
{
    //Declare a pointer of class type
      myclass* ptr, *ptr1;
    // Allocate dynamically
      ptr = new myclass (6, 7);
        └→ Parameterized constructor
                    called.
      ptr1 = new myclass();
        └→ ~~param~~ default construc-
                tor called.
      // output section
        ptr → display();
        ptr1 → display();
        return 0;
  } // end of main()
```

<u>Output:-</u>

```
6  7
0  0
```

Lecturenotes.in

3.9.10

Destructors :-

→ A special member function use to destroy the object when it goes out of scope.

→ Its name is same as class name.

→ Takes no arguement

→ Declaration always precede with tilde (~) sign.

eg

```
// class defination
class myclass
{
    int x,y;
        public:
            myclass()
            {
                x = 10;
                y = 20;
            }
```

Lecturenotes.in

```
~ myclass ()
  {
     cout<<"\n object destroyed";
  }
};
// end of class defination
// start of main
int main()
{
   // Declaration section
   myclass obj;
   {
      myclass ob1;
   } → Destructor called
                for ob1 when
   return 0;      block ends
}; → Destructor called for obj.
// end of main
```

<u>Output</u>:- Object destroyed.

Lecturenotes.in

Friend function:-

→ It is not in the scope of a class to which it is friend. It is not the memberfunction of the class but can acces the members (private and public) of a class to which it has been declared as a friend.

→ Syntax:-

```
friend rtype fun name(argument);
```

↑

This declaration has to be written inside the class to which it is declared as a friend.

Lecturenotes.in

```
// Defination of friend function

 float average() // float average
 {                      (data
                         obj)
        float x;
        data obj(10,11); // not needed
        x = (obj.var + obj.var1)/20
                                  ;
       return x;
  }
 // end of defination

 // start of main()

  int main()
  {
        ~~float~~ //Declaration section
         float y;
        //data obj1(20,30);
```

Lecturenotes.in

```
    // function call

        y= average(); // y= average(obj1)

    // output section

        cout<<"\n Average ="<<y;

        return 0;

  } // end of main
```

Properties of friend function:-

→ Friend function is not in the scope of the class to which it has been declared as friend.

→ It is invoked like a normal function.

→ It cannot access the members directly (It needs a object to access the members).

LectureNotes.in

→ It can access all private and public members of a class.

→ Usually it has the object as argument.

→ A function can be friend of a single class.

→ A function can become friend of more than class.

→ A member function of one class can become friend of another class.

→ A class can become friend of another class.

Lecturenotes.in

6.9.10

```
// class defination
class myclass
{
        int x, y;
```

```
public:
    void setdata(int,int);
   friend int mean(myclass);
};
// end of class
// ~~some~~ functions definiation
void myclass:: setdata(int a,
                        int b)
 {
       x=a;
       y=b;
 }
int (mean(myclass obj)
 {
     return((obj. x+obj. y)/2);
 }
// end of function defination
```

LectureNotes.in

```
// start of main
int main()
{
      // Decleration section
    myclass obj1

      // Input section
        obj1.setdata(5,6);

      // output section
      cout<<"\n Mean="<<mean(obj1)
      return 0;
}
// end of main
```

→ A friend function can be a friend to more than one class

Lecturenotes.in

eg

```
class myclass2; // forward
// class defination  declaration
      of myclass1.
class myclass1
{
      int x;
      public:
        void set~~data~~x(int);
       friend int max(myclass1,
                         myclass2);
};
// end of defination
```

During the compilation of this statement will give myclass2 no meaning because it is a user-defined d.type. So to avoid confustion of compiler we will give a forward declaration of myclass 2.

Lecturenotes.in

```
// class defination of myclass2

class myclass2
{
        int y;
      public:
          void sety(int);
     friend int max(myclass1,
                      myclass2);
}; end of class defination
```

//There is no problem if this function appeare in public or private ~~p~~ sections of a class. It can appear in any section because it is not a member function of class and thus accessibility rules don't apply to it.

Lecturenotes.in

```
// member fun^n defination

void myclass1::setx(int a)
{
        x = a;
 }

 void myclass2::sety(int b)
  {
        y = b;
   }
// end of defination

 // friend function defination

 int max(myclass1 obj1, myclass2
                         obj2)
 {
        int max1;
        if(obj1.x > obj2.y)
           max1 = obj1.x;
```

Lecturenotes.in

```
else
    max1= obj2.y;
    return max1;
}
//Start of main()
int main()
{
    // Declaration section
     myclass1 obj1;
     myclass2 obj2;
    // Input section
       Obj1. setx(5);
       obj2. sety(6);
    // Output section
       cout<<"\n Maximum is:";
```

Lecturenotes.in

```
    cout << max(obj1,obj2);
    reteon 0;
}
// end of main()
```

→ we can not overload a friend function.

→ A friend function of one class can be member to another class.

eg

```
        // forward declaration of
            myclass 1
        class myclass1;
        class myclass
        {
            int x,y;
          public:
            void setdata(int, int);
            void display(myclass1);
        }
        // end of class
```

Lecturenotes.in

```
// member fun's defination

void ~~set~~ myclass::setdata(int a,
                              int b)
{

        x = a;
        y = b;
}

void myclass::display(myclass1.
                              obj)
{

     cout<<"\n Members of
                 myclass are:";
       cout << x <<"and"<< y;
    cout<<"\n Member of
               myclass1 ~~are~~ is:";
     cout << obj.var;
}
// end of defination
```

Lecture Notes .in
Lecturenotes.in

```
// defination of myclass1

class myclass1
{
        int var;
      public:
          void  setdata(int x)
          {
                    var = x;
          }
    friend void myclass::display
                            (myclass1);
};

// end of defination
// start of main
 int main()
 {
        // Declaration section
             myclass obj;
```

Lecturenotes.in

```
myclass1 obj1;
//input section
    obj. setdata(10, 20);
    obj1.setdata(30);
//calling friend-function
    obj1. display(obj);
   return 0;
 }
 // end of class
```

Friend class:-

```
class myclass1;
class myclass
{
    int x, y;
  public:
     void setdata(int, int);
     void display(myclass1);
};
```

LectureNotes.in

```
class myclass1
{
      int var;
    public:
      void setdata(int x)
       {
           var = x;
       }
   friend class myclass;
};
```

13.9.10.

Operator Overloading:-

Changing the operand types (i.e; using the objects as operands).

eg- $\underbrace{5 + 6}_{\text{Integer type (operands)}}$

LectureNotes.in

$$\underbrace{3.6 + 7.2}_{\text{float type (operands)}}$$

$$\underbrace{a + b.}_{\text{char type (takes their associativity)}}$$

$$\underbrace{Obj1 + Obj2}_{\text{objects of same class}}$$

Lecturenotes.in

⇒ This is called as overloading.

When an operator is overloaded, the meaning of the operator is not changed rather the operator is assigned ~~is assigned~~ with some extra functionality (work) by using the user defined types. (class) as operand.

Procedure for operator overloading

→ The user has to define the procedure of manipulating the user defined type.

→ The task is accomplished by a special function called as operator function.

→ ~~The task is accomplished by a special function called as operator function.~~

→ The operator function can be

a) member function

b) non-member (friend) fun

Lecturenotes.in

When the operator function is member function:-

The prototype of operator function is

Syntax:-

r-type operator # (arglist);

return type

keyword used to specify the operator function

place holder which will be replaced by an operator

eg→ To overload binary operator '+'.

The prototype of corresponding operator function is

Lecturenotes.in

class name operator + (arg list);

<u>or</u>

class name operator + (classname

<u>Note</u>:-

→ when we are overloading a binary operator and operator function is a member function then only right operand ~~is~~ will pass to the operator function as argument.

→ the left operand is responsible to Envoke the operator function i.e; it is passed to the operator function by <u>this pointer</u>.

Lecturenotes.in

When the operator function is a member function - defination:-

Syntax:-

```
r-type classname::operator#
                (arglist)
{
    function body
    - - -
    - - -
}
```

eg

```
// overload binary (+) for
a class loc.
// start of program
# include <iostream>
using namespace std;
```

Lecture Notes .in
lecturenotes.in

```
// class defination starts
 class loc
{
      int longitude;
      int lattitude;
    public:
       loc() // Default constru-
                          ctor
       {
          longitude=0;
          lattitude=0;
       }
     loc(int lt, int la)
          // parameterized
                constructor
     {
          ~~lat~~
          longitude=lt;
          Lattitude=la;
     }
```

Lecturenotes.in

```
// Prototype declaration
        for operator overloading

  loc operator +(loc);

 void show()
 {
      cout<< longitude << endl;
      cout << lattitude;

 }

};

// end of class defination

// Defination of operator function

 loc loc:: operator +(loc obj)
 {
      loc tempobj;
      tempobj. longitude =
           obj. longitude +
             longitude;
```

Lecture Notes .in

```
  tempobj. lattitude = obj. lattitude
                + lattitude;

  return tempobj;

}
// end of defination
// start of main
 int main()
 {
      // input section
        loc obj;
        ↳ caling default
                constructor.
      loc. obj1(10, 11);
      loc. obj2(20, 30);
        ↳ calling parameter-
               ized construtor
```

Lecture Notes .in
Lecturenotes.in

```
// calling operator fun^n

  obj= obj1+obj2;

 // output section

  obj.show()

  return 0;

 } // end of main

// end of program.
```

Output

```
    30

    41.
```

16.9.10

Q- wap to overload binary '-' operator.

Lecturenotes.in

```
// start of program
 #include<iostream>
 using namespace std;
// class defination
  class loc
  {
      int longitude;
      int latitude;
    public:
      loc()
      {
          longitude=0;
          latitude=0;
      }
      loc(int lt, int la)
      {
          longitude=lt;
          latitude=la;
      }
```

Lecturenotes.in

```
void show()
{
     cout<<"\n "<<longitude;
     cout<<latitude<<endl;

   }

  loc operator -(loc);

};
// end of class defination
// Defination to overload -

loc loc:: operator-(loc obj)
{
        loc temp;
        temp.longitude =
               longitude -
                     obj.Longitude;
```

Lecturenotes.in

Q- wap to overload '=' operator.

```
// start of program
#include<iostream>
 using namespace std;
// start of class
 class loc
 {
      int longitude;
      int latitude;
    public:
      loc()
      {
         longitude=0;
         latitude=0;
      }
```

Lecturenotes.in

```
    loc(int lt, int la)
    {
          longitude = lt;
          latitude = la;
     }
    void show()
     {
          cout<<"\n"<<longitude;
           cout<<endl<<latitude;
      }
    void operator = (loc);
 };
// end of class.
// defination of fun^n
void loc::operator=(loc obj)
{
    longitude = obj.longitude;
```

Lecturenotes.in

latitude = obj.latitude

↓

with this statement multiple assignment ~~operato state~~ will not work i.e, ~~obj~~4 = obj = obj1.

```
} // end of def^n of fun^n
// start of main
int main()
{
    // Declaration section
    loc obj, obj1(35,36);
    loc obj2(10,11);
    // calling function
    obj = obj1;
```

Lecture Notes .in
Lecturenotes.in

```
    // output section
      obj. show();
    return 0;
 } // end of main
 // end of program
```

Q- wap to overload unary ++ operator.

→ when we are overloading of unary operator either prefix or postfix and the <u>operator function is a member function.</u> Then no argument is passed to the operator function to overload prefix operator and one dummy integer argument

LearnNotes.in

is passed to the operator function to overload postfix operator.

Syntax:-

a) To overload prefix(increment or decrement operator).

```
Rtype operator # ()
{

}
```

where # is the place holder for '++' or '--'

b) To overload postfix(increment or decrement operator).

Lecturenotes.in

```
Rtype operator#(int)
{                  ↑
                 dummy
                   variable.

}
```

Here the operator function is a member function.

```
→ //start of program
  #include <iostream>
  using namespace std;
  // class defination
  class loc
  {
        int longitude;
        int latitude;
     public:
        loc ()
        {
           longitude=0;
             latitude=0;
        }
```

Lecturenotes.in

```
loc( int lt, int la)
{
    longitude = lt;
    latitude = la;
}
void show()
 {
     cout<<"\n"<< longitude;
     cout<<"\n"<<latitude;
 }

 void operator++( );
//void operator ++(int);
}; // end of defination
// defination of fun^n
void loc:: koperator++(int)
 {
     ++ longitude; // longitude++;
     ++ latitude; // latitude++;
 }
```

Lecture Notes .in
Lecturenotes.in

```
// start of main()
 int main()
{
      // Declaration section
      loc obj(35, 36);
      //  o/p before fun^n call
       obj. show()

       // Fun^n call
          ++obj;//obj++;
        //o/p after fun^n call
          obj.show();
          return 0;
  } // end of main
 // end of program
```

Lecturenotes.in

Assignment:-

Q wap to overload postfix and prefix decreement unary operatr.

```
// start of program.
# include <iostream>
 using namespace std;
// class defn
 class loc
 {
     int longitude;
     int latitude;
   public:
     loc()
     {
         longitude=0;
         latitude=0;
     }
```

Lecturenotes.in

```
    loc(int lt, int la)
    {
        longitude = lt;
         latitude = la;

     }
     void show()
     {
        cout<<"\n"<<longitude;
        cout<<"\n"<<latitude;
     }
     void operator--();
     void operator--(int);
  };
  // end of class
 // defination of fun^n
 void loc:: operator--()
 {
        --longitude;
         --latitude;
  }
```

Lecturenotes.in

```
void loc:: operator--(int)
 {
        longitude--;
         latitude--;

  }
//end of fun^n defination
// start of main
int main()
 {
      // Declaration section
        loc obj(8, 10);
     // output before any fun^n call
         obj. show();
     // fun^n call for prefix
         --obj;
```

Lecturenotes.in

```
//show after fun^n call
    obj.show();
  // fun^n call for postfix
    obj--;
  // calling show after fun^n call
    obj.show();
   return 0;
} // end of main
 // end of program
```

Q- wap to overload the short hand operator (+=, -=).

Lecturenotes.in

//

~~class loc;~~

{

~~tot t.~~

```
class loc
{
      int longitude;
      int latitude;
   public:
      loc()
     {
        longitude=0;
         latitude=0;
     }
      loc(int lt, int la)
     {
         longitude=lt;
          latitude=la;
      }
   void show()
   {
        cout<< longitude";
        cout<< latitude;
   }
```

Lecturenotes.in

```
~~void~~ loc operator +=(loc obj)
{
        ~~set~~
        longitude += obj.longitude;
        latitude += obj.latitude)
}

loc operator -=(loc obj)
{
        longitude -= obj.~~latitude~~ longitude;
        latitude -= obj.latitude;
 }
};
int main()
{
        loc obj;
        loc obj1(10,12);
        loc obj2(11,15);
        ~~obj1 ——~~ obj2+=obj1;
        obj2.show();
        obj2-=obj1;
        obj2.show();
        return 0;
}
```

Lecturenotes.in

17.9.10

Some Restrictions with Operator overloading :-

→ The no. of operands for a operator cannot be changed.

→ The precedency and associativity of an operator cannot be changed.

→ we cannot overload operators like,

a) :: (scope Resolution operator)

b) • (Dot operator / member accessing operator)

c) .* (member accessing operator).

d) ?: (conditional or ternary operator)

Overloading using friend function

→ friend function as operator function.

- Overloading binary operator
  → The operator function will have two arguments.

- overloading unary operator
  → The operator function will have one argument.

eg overload binary '+' using friend function

General Syntax:-

```
(classtype)
r-type operator #(classtype, classt
```

Lecturenotes.in

```
// start of program
#include<iostream>
using namespace std;
// class defination
class loc
{
        int longetude;
        int latitude;
    public:
        loc();
        loc(int,int);
        friend loc operator+(loc
                              loc);
        void show();
};
// end of class
```

Lecturenotes.in

```
// member function defination
 loc :: loc()
 {
          longitude=0;
          latitude=0;

 }

 loc:: loc(int a, int b)
 {
          longitude=a;
          latitude=b;
 }
          ~~loo~~ loc
 ~~friend~~ ~~loc~~: operator +
                   (loc obj, loc obj1)
 {
          loc temp;
          temp. latitude =
               obj. latitude +
                 obj1. latitude;
```

Lecturenotes.in

```
    temp.longitude = obj.longi
               tude + obj1.longitude

   return(temp);
}

void loc::show()
{
     cout<<"\n longitude"<<longitu
    cout<<"\n latitude"<<latitude

 }
// end of definations

// start of main
 int main()
 {
       // Declaration section

        loc obj, obj1(10,20);
         loc obj2(30,40);

       // function call
         obj = obj1+obj2;
```

Lecturenotes.in

```
    // function call for output
        obj.show();
        obj1.show();
        obj2.show();
        return0;
 }// end of main
 // end of program
```

5- Overload unary ++ using friend function

```
// start of program
#include<iostream>
using namespace std;
// class defination
class loc
{
        int latitude;
        int longitude;
    public:
        loc()
```

LectureNotes.in

```
{
        longitude =0;
         latitude =0;
  }
 loc(int lt, int la)
   {
        longitude = lt;
        latitude = la;
   }
 friend loc operator++
                   (loc);
 void show()
  {
       cout<<"\n"<<longitude;
       cout<<"\n"<<latitude;

  }
};
//end of defination
```

Lecturenotes.in

```
// Fun^n def^n          But if we use
                        then it will
loc operator ++(loc obj)
                                 // ++ ob
{
        loc temp;   // ++ obj.
        temp.latitude = ++ ob
                           lat
        temp.longitude = ++ o
                           lon
        return temp;
} // end of defination

// start of main
int main()
{
        // Declaration Sectio
        loc obj, obj1(1920);
        loc obj2(30,40);

        // function call
        obj = ++obj1; // ++
```

If we write this, then it will not work because obj will is local to this function only

Lecturenotes.in

```
// output section
    obj.show();
     obj1.show();
     obj2.show();
    return 0;
} // end of main
// end of program.
```

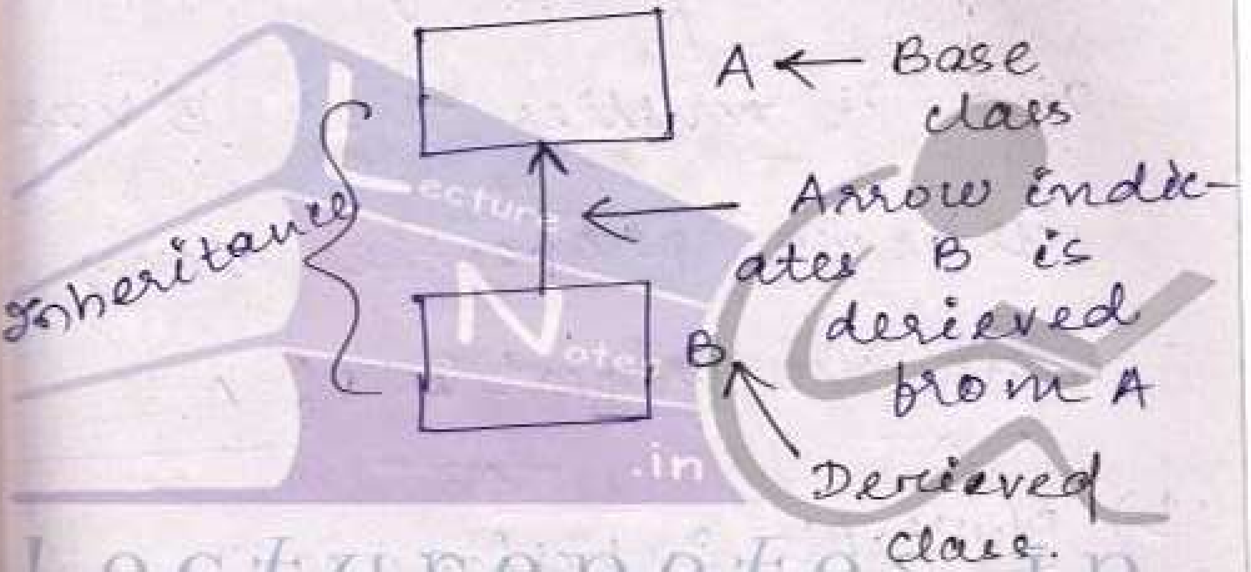
Lecturenotes.in

20.9.10

INHERITANCE.

It is the proces by which a class acquires all the properties of other classes.

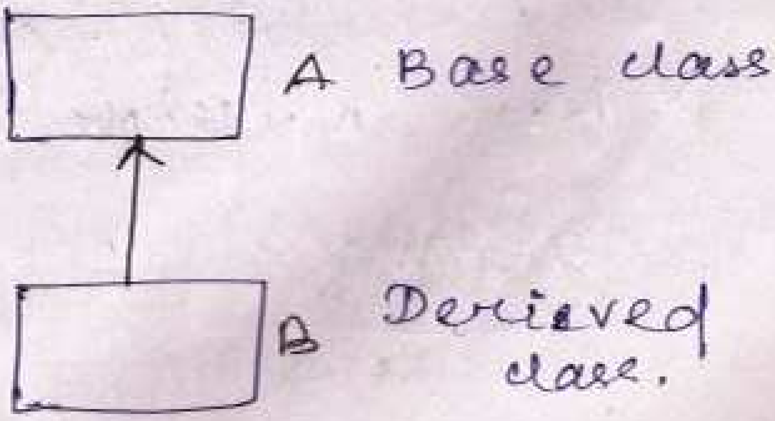
eg→ class A, class B



Lecturenotes.in

Ways of Inheritance:-

a) Single Inheritance:-



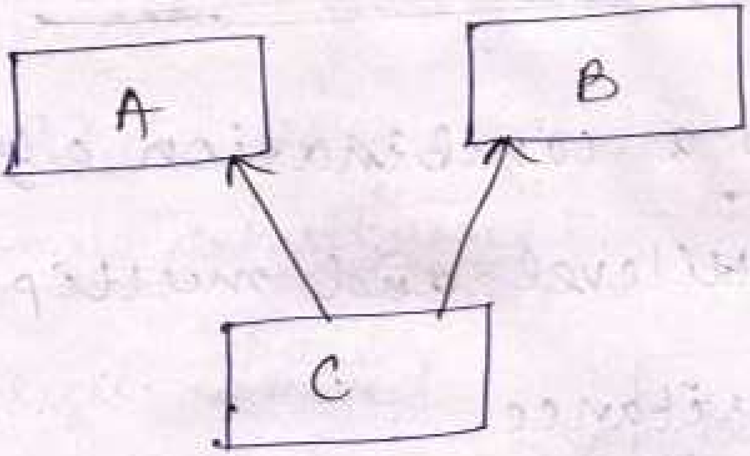
→It has one base class and its properties are acquired by one derieved class.

→The arrow indicates that B is derieved from A and it implies that B has its own properties along with the properties of A.

b) Multiple Inheritance.

→A single derieved class acquires the properties from multiple base class.

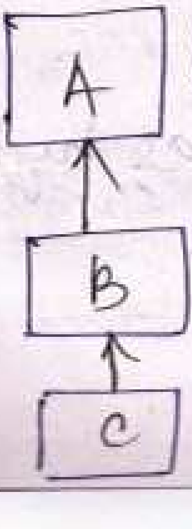
LectureNotes.in



→ Derived class C has its own properties + properties of A + properties of B.

c) Multi-level inheritance :-

→ A class derived from its bas class becomes the base class ~~from~~ for some other class



(Only properties o A)

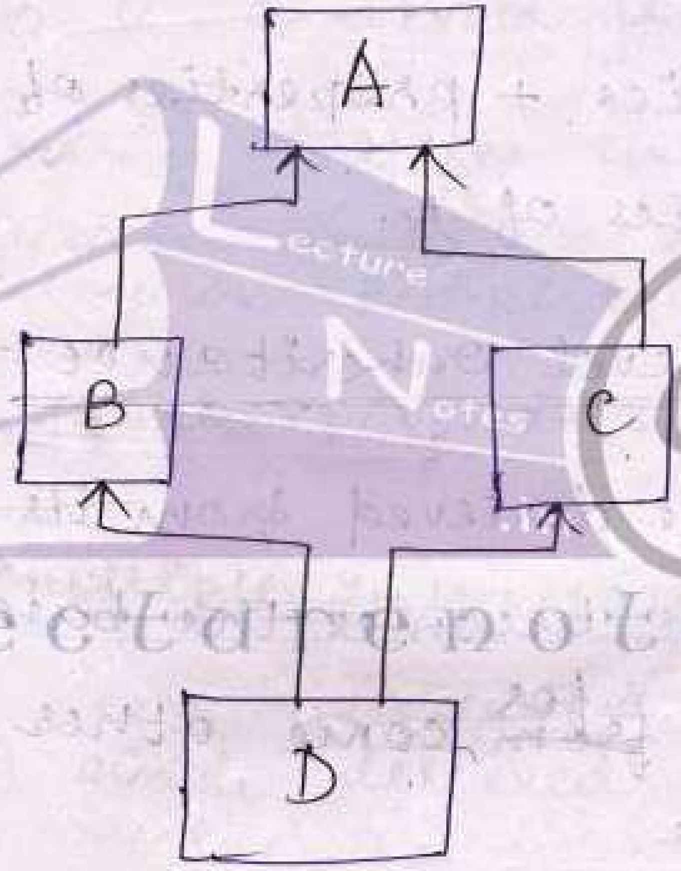
Properties of A +

Properties of A+B

lecturenotes.in

d) Hierarchial Inheritance:-

→It is a combination of single, multilevel and multiple inheritance.



→Also called as hybrid inheritance.

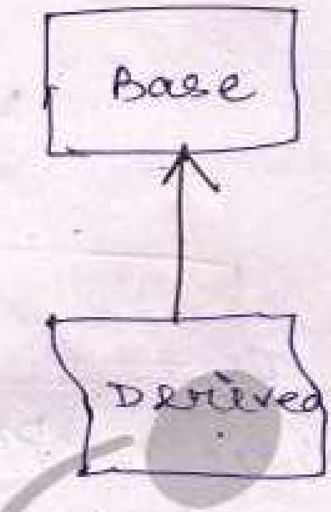
Lecturenotes.in

Implementation of Inheritance:-

Syntax:-

1) **single Inheritance:-**

```
class ~~derived~~ base
{
      public:



  };
```

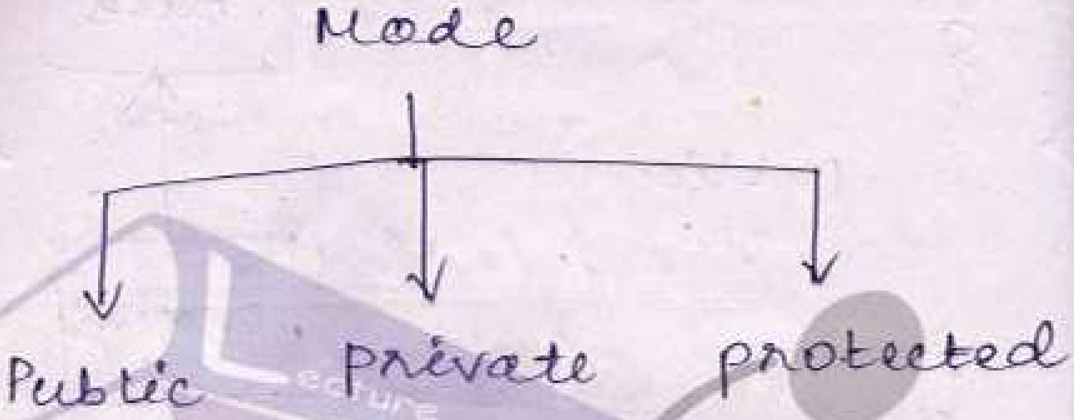


```
class derived : mode base
 {

        public:




    };
```

Lecture Notes .in
Lecturenotes.in

Mode → Defines the mode of inheritance (ie. it

Specifies the way base class members are accessed by derived class).



1) Public mode →

→ All public members of base class becomes public members of derived class

→ All protected members of base class becomes protected in derived class.

Lecturenotes.in

→ All private members of base class are never accessed by the derieved class members directly.

Protected Access Specifier:-

The protected members (the members which appears unde protected access specifier) of class are only visible inside the derieved class (i.e the derieved class can access the protected members directly.

2) Protected mode:-

→ All public and protected members of base class becomes protected member of derieved class.

lecturenotes.in

3) Private mode:-

All public and protected members of base class become private of derieved class.

eg of single inheritance:-

```
// Base class defination
class base
{
      int i, j;
   Public:
      void set( int x, int y)
      {
            i= x;
            j= y;
       }
```

Lecturenotes.in

```
void show()
 {
     cout<<"\n Base members";
     cout<<i<<j;
  }
 };
//Derieved class defination
class derieved: public base
 {
     int k;
   public:
     void setk(int x)
     {
           k = x; //set(x+2,
                        x+3);
     }
     void showk()
     {
         cout<<"\n Derieved
                    members";
         cout<<k;
     }
 };
```

Lecture Notes .in
Lecturenotes.in

// start of main()

```
int main()
{
        derieved ob;

        ob.set(5, 6);
```

↑ since mode is public so ob can access the set function of base class.

```
        ob.show();
```

↑ since show is a public member of base class and mode is public.

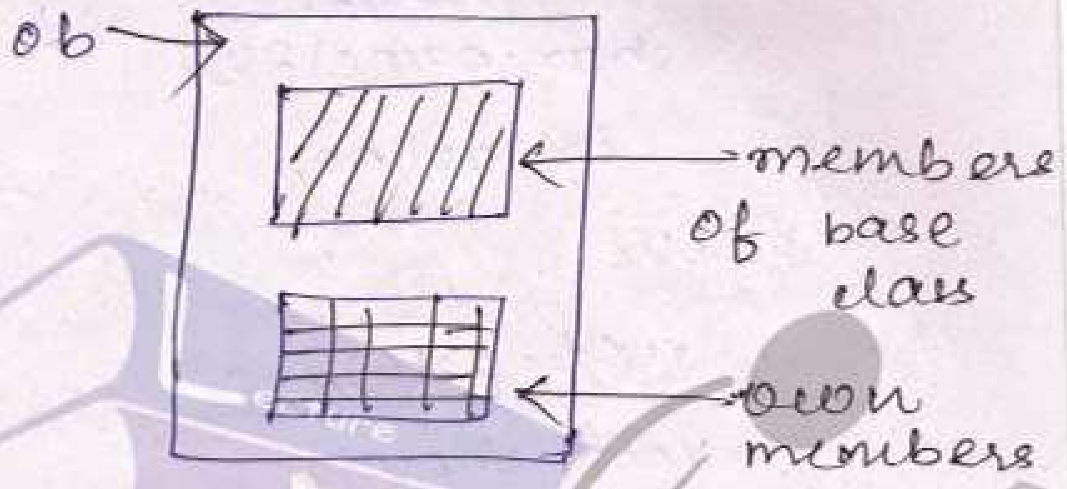
```
        ob.setk(10);
        ob.showk();
```

→ calling its own functions

Lecturenotes.in

```
return 0;
}
// end of main
```

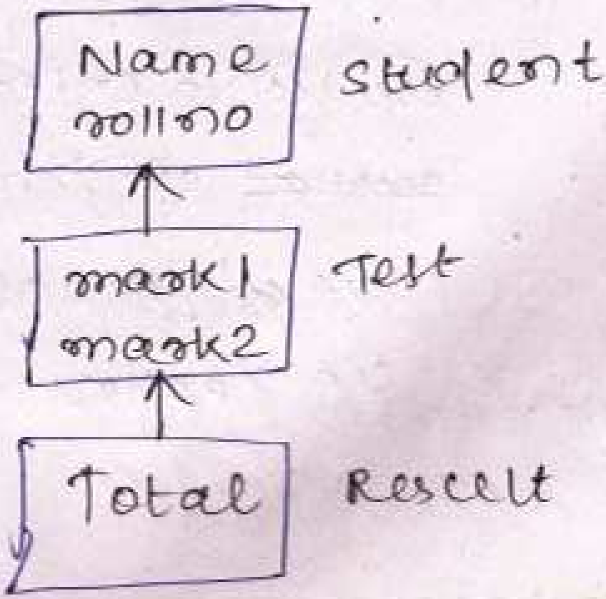
Execution:-



Benefit of Inheritance.

23.9.10

Program to show multilevel inheritance:-



```
// ~~class~~ base class defination

 class student
 {
     protected:
            char name[20];
              int rollno;
      public:
            void get();
            void show();
   }; // end of base class
                    def^ns
// member fun^ns ^ of base class

  void student::get()
  {
        cout<<"\n enter name";
        ~~cout<<"~~ cin>>name;
        cout<<"\n enter rollno";
          cin>>rollno;
  }
```

Lecturenotes.in

```
void student::show()
{
        cout<<"\n Members of class
                     student :";
          cout<<"\n Name:"<<name;
         cout<<"\n Roll no"<<rollno;

}

// derieved class Test defination

class test: public student
{
        protected:
            int mark1;
            int mark2;
        public:
            void getmarks()
            {
                cout<<"\n Enter mark1
                  cin>>mark1;
                cout<<"\n Enter mark2:
                   cin>>mark2;
            }
```

lecturenotes.in

```
void showmarks()
 {
      cout<<"\n Testmarks";
      cout<<mark1;
       cout<<mark2;
   }
 }; //end of derived class Test

//class Result defenation
  class result:public test
 {
      private:
        int total;
     public:
       void display()
       {
           total=mark1+
                    mark2;
            show();
           showmarks();
            cout<<"Total
                  marks"<<total
       }
```

Lecturenotes.in

```
        cout << total;
    }
}; // end of class Test.

// start of main
int main ()
 {
          //Implementation of
                        hierarchy

          // Declaration section

            result obj;

          // input section

             obj.get();
             obj.getmarks();

          // Display section

             obj.display();

         return 0;
 } // end of main
```

Lecturenotes.in

Implementing Multiple Inheritanc

```
//defination of base1
class Base1
{
      protected:
          int x;
      public:
         void get()
         {
              cout<<"\n enter x";
              cin>>x;
         }
         void show()
         {
              cout<<"\n Members
                      of base";
              cout<<x;
         }
};
// end of defination of Base1
```

Lecturenotes.in

//defination of Base2

~~class derieved: public Base1,~~
~~public Base2~~

```
class Base2
{
    protected:
        int y;
    public:
        void get()
        {
            cout<<"\n enter y
            cin>>y;
        }
        void show()
        {
            cout<<"\n member
                    of base2";
            cout<< y;
        }
};
// end of defination of Bas
```

lecturenotes.in

```
// defination of derieved
class derieved: public Base1,
                        public Base2
{
    private:
        int z;
    public:
        void getz()
        {
            cin>>z;
        }
        void showz()
        {
            cout<<z;
        }
};
// end of derieved class
```

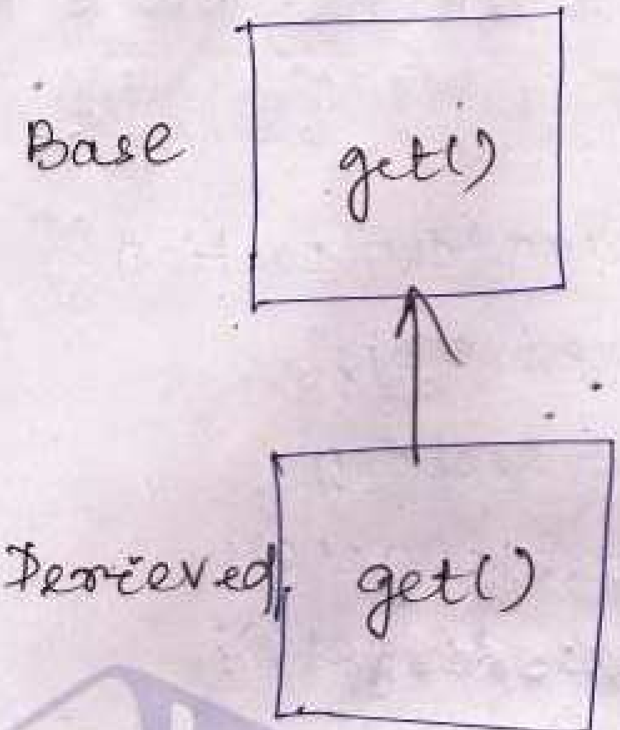
Lecturenotes.in

```
// start of main
int main()
{
    // declaration section
    derieved obj;
    // input section
    obj. Base1::get();
    obj. Base2::get();
    obj. getz();
```

we avoid ambeguity by using the classname to call the member functions of the base classes if they have same name

```
    // output section
    obj. Base1::show();
    obj. Base2:: show();
    obj.showz();
    return 0;
} // end of main
```

lecturenotes.in



If we write,

~~obj get~~

derieved obj ~~get()~~;

obj.get()

→ It will invoke the member fun of get() of derieved class because of higher priority

Lecturenotes.in

function overriding:-

Redefining a base class ~~defination~~ function inside the derived class with a new defination is called as function overriding.

eg.

```
// Base class defination
class Base
{
    public:
        void show()
        {
            cout<<"\n Inside
                      base class";
        }
};
// end of base class defination
```

Lecturenotes.in

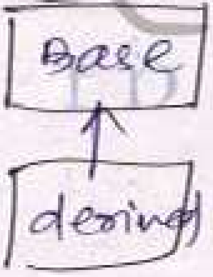
```
// Derieved class defination

class derieved : public Base
{
    public:
        void show()
        {
            // base::show()- Statement (1)
            cout<<"\n inside
                  derieved class";
        }
        Function Overriding
};
// end of derieved class

// Start of main()

int main()
{
        // Declaration section

            derieved obj;
```

Lecturenotes.in

```
// calling show() of derieved
                        class

    obj.show();

  // calling show() of base cla

    obj base::show();
  ↳// No need to write if we
      include statement 1
   return 0;

 }
// end of main
```

<u>Inheritance & constructor:-</u>

→ // Base class defenation (single)



```
class Base
{
    public:
       Base()
       {
           cout<<"\n Base class
                    constructor";
       } ~Base() { cout<<"\n Base"}
};
//end of base class
```

Lecturenotes.in

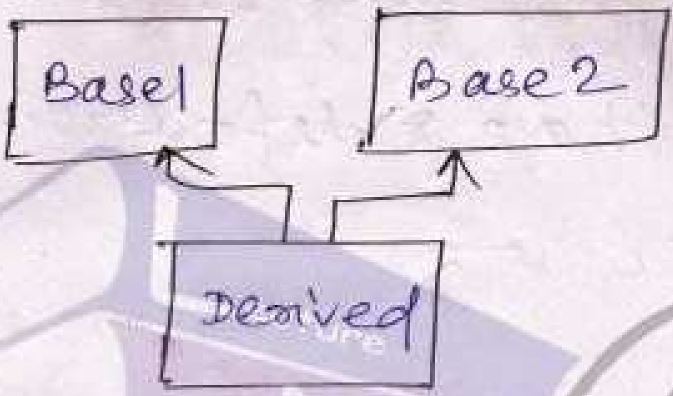
```
// derieved class defination
 class derieved: public Base
 {
    public:
         derieved()
        {
           cout<<"\n Derived
                      constructor";
        }.
      ~derieved(){cout<<"Derived"
 };
 //end of derieved class
 //start of main
  int main()
  {
        derieved obj;
         return 0;
   }
  // end of main
```

LectureNotes.in

Output

```
Base class constructor
Derived constructor
Derived
 Base.
```

→ (Multiple)



```
// class base1 definition

class base1
{
      public:
          base1()
         {
           cout<<"\n Base1
                    Constructor";
         }.
```

Lecturenotes.in

```
~base1()
  {
       cout<<"\n Destroying base1
  }
};// end of base1
// class base 2 defenation
class base2
{
      public:
         base2()
         {
              cout<<"\n Base2
                         constructor";
         }
       ~base2()
         {
              cout <<"\n Destroying
```

Lecture Notes
Lecturenotes.in

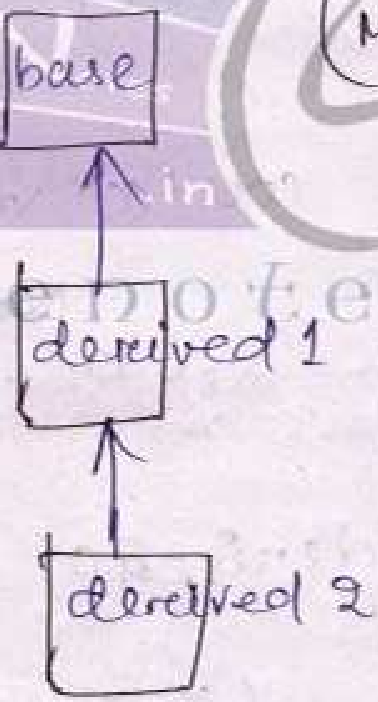
```
        base 2";
    }
};// end of base 2.
// derived class defination
class derived: public base1,
                  public base 2.
{
    public:
        derived()
        {
            cout<<"\n Derived
                       constructor";
        }
        ~derived()
        {
            cout<<"\n Destroying
                       derived";
        }
};
```

Lecturenotes.in

Output :-

```
Base1 constructor
 Base 2 constructor
 Derived constructor

 Destroying derived
 Destroying base 2
 Destroying base 1
```

→ (Multilevel)



Lecturenotes.in

```
//Base class defination
class base
{
    public:
        base()
        {
            cout<<"\n Base constructor";
        }
        ~base()
        {
            cout<<"\n Destroying base";
        }
};
//end of base class
//derived1 class defination
class derived1 : public base
{
    public:
```

Lecturenotes.in

```
derived1()
{
   cout<<"\n Derieved 1
        constructor";
 }
~derived1()
 {
    cout<<"\n Destroying
          derived 1";
  }
};//end of class derived1
//derived2 defination
class derived2: public derived1
{
      public:
      derived2()
       {
```
Lecturenotes.in

```
    cout<<"\n Derieved 2
            constructor";
  }
  ~derived 2()
   {
      cout<<"\n Destroying
                derived2";
     }
 }; // end of derived 2
// start of main
 int main()
 {
       derived 2 obj;
         return 0;
   }
 // end of main
```

Lecturenotes.in

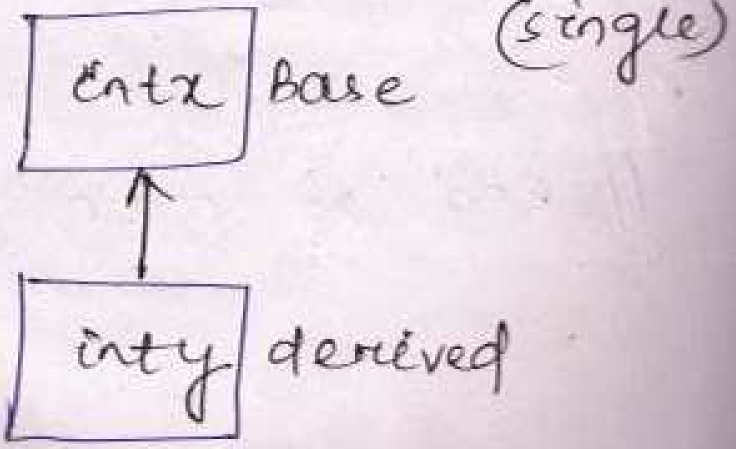
Output :

```
Base constructor.
Derived1 constructor
Derieved2 constructor
Destroying Derived2
Destroying Derived1
Destroying Base
```

Parameterised constructor :

When a base class contains parameterised constructor, its mandatory to define parameterised constructor in derived class.

→



Lecturenotes.in

eg:-

```
// Base class defenation
class Base
{
      protected:
            int x;
      public:
         Base()
         {
              cout<<"\n Base class
                    default constrcu-
                              ctor";
         }
         Base(int t)
         {
                x = t;
          }
          ~Base()
          {
              cout<<"\n Destroying
                            base";
           }
   }; // end of defination of base
```

Lecturenotes.in

```
// derived class defination
 class derived: public Base
 {
        int y; //int z;
        public:
            derived()
             {
                  cout<<"\n derived
                          class default
                       conctructor";
             }
            derived(int a, int b):
             {                Base( b)
```

argument is passed to the parameterized constructor of base class.

Lecturenotes.in

```
            y=a; //z=b.
    }
    ~derived()
     {
         cout<<"\n Destroying
                         derived";
     }
 };  void show()
       {
           cout<<x<<endl;
           cout<<y;
           cout<<"\n"<<z;
       }
 }; // end of defination of derived
                                class
 // start of main
    int main()
    {
         // Declaration section
            derived obj(10,20);
```

LectureNotes.in

```
// Display section
    obj.show();
    return 0;
 }// end of main
```

<u>Output</u>:-

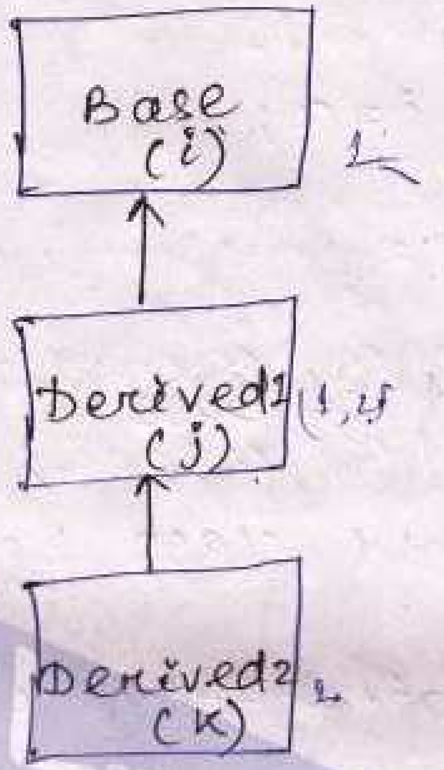
~~10~~
~~20~~
~~20~~
~~Base class de~~

```
10
20
20
Destroying Base
Destroying Derived
```

Lecturenotes.in

24.9.10 :

→ (multilevel)



Lecturenotes.in

```
// Base class defination
class Base
{
    protected:
        int i;
    public:
        Base()
        {
            cout<<"\n Inside Base ";

        }
    Base(int x)
     {
           i=x;

      }

  }; // end of Base

 // Derived1 class begins

  class derived1: public Base
  {
        prootected:
            int j;
        public:
            derived1()
            {
                 cout<<"\n Inside
                           derived1";
            }
```

Lecturenotes.in

```
   derived1(int x, int y): base(y)
   {
        j= x;

    }
 }; // end of derived1.

// derived 2 class begins

  class derived 2: public derived1
  {
        int k;
      public:
         derived2()
         {
             cout<<"\n inside
                         derived2";
         }

       derived2(int x, int y, int z):
              derived1(y,z)
       {
           k= x;
       } void show();
  }; // end of derived 2
```

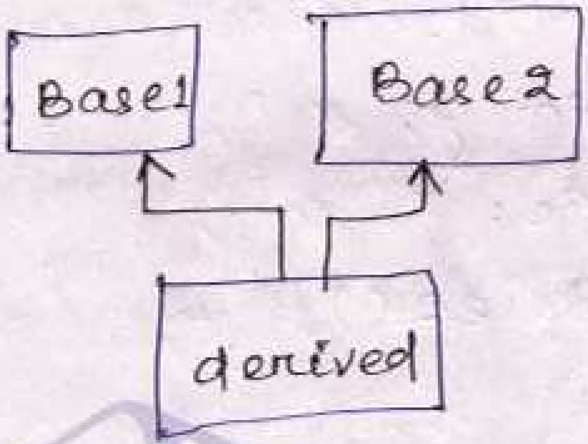
Lecturenotes.in

```
// show defination
 void derived2::show()
 {
       cout<<"\n Members are:";
        cout << i<<endl;
       cout << j<<endl;
       cout<< k << endl;
  } // end of show

//start of main
int main()
{
   // Decleration Section
       int x, y, z ;
      cout<<"\n Enter x, y, z:";
       cin >>x >>y>>z;
      derived2 obj(x, y, z);

      // Display section
         obj.show();
```

Lecturenotes.in

```
    return 0;
} // end of main
```

→ (Multiple)



```
// Base1 defination
  class Base1
  {
     protected:
        int i;
     public:
        Base1()
        {
           cout<<"\n Inside Base1";
        }
       Base1(int x)
       {
             i=x;
       }
  }; // end of Base1
```

Lecturenotes.in

```
// Base2 defination
 class Base2
 {
      protected:
          int y;
      public:
          Base2()
          {
             cout<<"\n Inside
                           Base2";
          }
          Base2(int z)
          {
                y=z;
          }
 }; // end of base 2
```

Lecturenotes.in

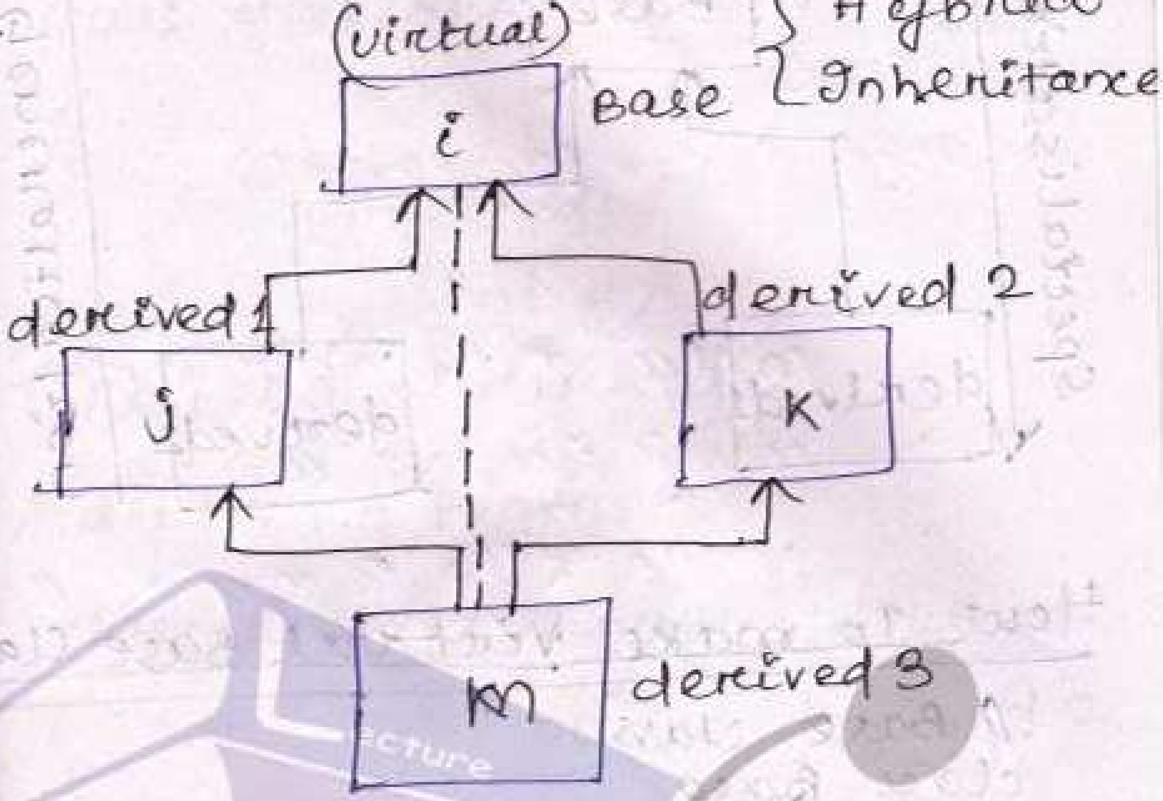
```
// derived class defination
class derived : public Base1,
                    public Base2.
{
     int k;
   public:
      derived()
      {
         cout<<"\n inside
                     derived";
      }
      derived(int x, int y, int z):
                  Base1(y), Base2(z)
      {
            k = x;
      }
      void show();
};
// end of derived class.
```

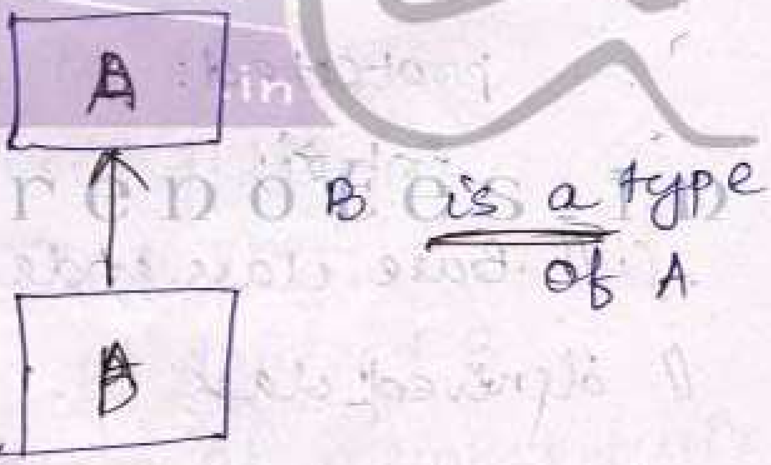
Lecturenotes.in

```
// show defination
void derived:: show()
{
     cout<<"\n Members are";
      cout<<" i= "<< i;
      cout<<" y= "<<y;
       cout<<" k= "<< k;
} // end of show

// start of main
 int main()
 {
        // Declaration section
        int x, y, z;
        cout<<"\n Enter x,y,z:";
        cin>>x>>y>>z;
        derived obj(x, y, z);
        // display section
          obj.show();
          return 0;
 } // end of main
```

Lecturenotes.in

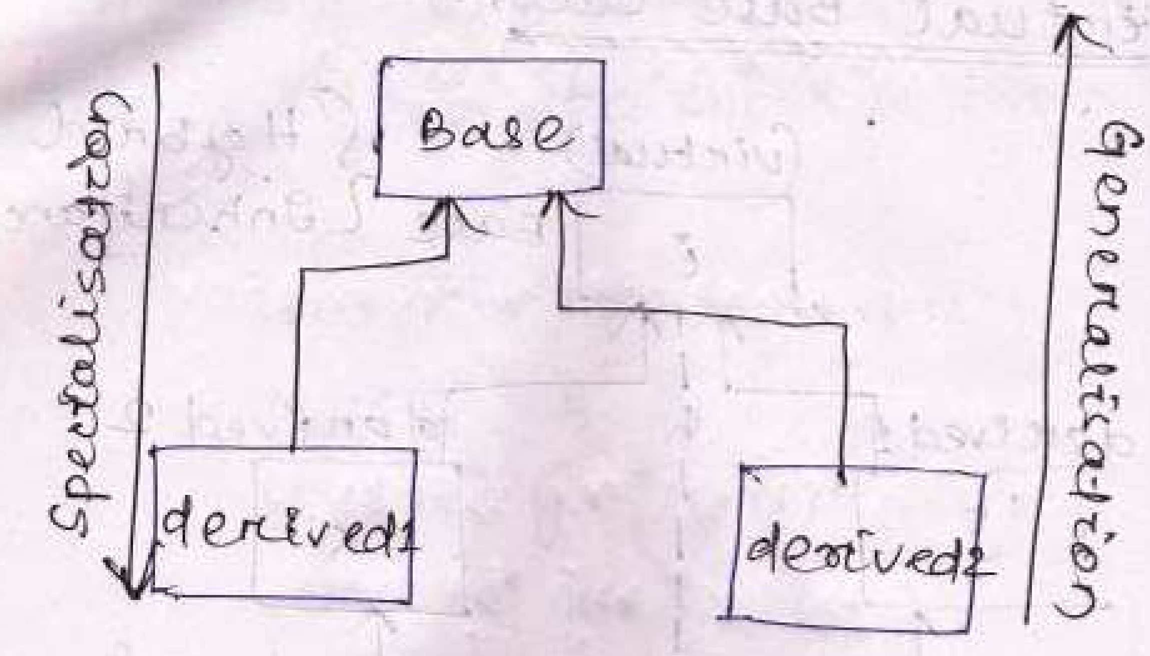
Virtual Base class:-





Inheritance is a ~~p.~~ type of relationship or hierarchy

part of → aggregation



How To make Virtual Base Class?

```
// Base class
class Base
{
        protected:
          int i;
 };// Base class ends

// derived1 class

class derived : public virtual
                             Base
{
          protected:
               int j;
 };// end of derived class
```

Lecturenotes.in

```
// derived2 class begins
 class derived2: public virtual
                         Base
 {
        protected:
             int k;
 }; // end of derived 2
// derived 3 begins
 class derived3: public derived1,
                   public derived 2
 {
        int m;
      public:
         void get();
         void show();
 }; // end of derived 3
// defination of member fun's of
                    derived3
 void derived3::get()
 {
       cout<<"\n Enter i, j, k
                    and m:";
```

Lecturenotes.in

```
        cin>>i>>j>>k>>m;
}
void derived3::show()
{
        cout<<"\n Members are:";
        cout<< i<<"\t"<< j;
        cout<<"\t"<< k <<"\t"<<m;
}
//end of definations
// start of main
int main()
{
        //Declaration section
          derived3 obj;
        // Input section
            obj.get();
```

Lecturenotes.in

```
// Output Section
   obj.show();
  return 0;

} // end of main
```

Objectives of Inheritance:-

→ Reusability
→ extendability.

→ If a class contains pointer as its members then we need to allocate memory for them in both default or parameterise so that memory is allocated at the time of creation and we don't get a segmentation fault.

→ Base obj; Derieved is Base type

Derieved ob;

Base
↑ public
Derieved

Obj = ob.
(Derieved properties truncated).

Object slicing

Lecturenotes.in

4.10.10.

Type Conversion:-

```
int x.
myclass obj;
myclass obj1;

obj = 5;

x = obj; X
```

↑ They are not of compatible data type.

LectureNotes.in

1) Built in type to class type.

Obj = x ;

(where x is built in, Obj is class type)

2) class type to built-in type

x = obj;

3) class type to class type

obj = obj1;

Built-in type to class type:

```
myclass obj(5,6);
```

type conversion
by using parameterised
constructor.

To convert one type to other type a special function is used, called the conversion function.

In this case, the conversion function, is a parameterised constructor.

Lecturenotes.in

eg:-

```
//class num begins
class num
{
      int x,y;
   public:
         num()
         {

         }
         num(int a)
         {
               x=a;
               y=a*a;
               cout<<"\n Inside
                          constructor"
         }
         void show();

};
// class num ends
```

Lecturenotes.in

```
// show defination.
 void num:: show()
 {
       cout<<"\n members";
       cout<<x<<y;
 }
// end of defination
// start of main
int main()
{
         //Declaration section
         int x=10;
         num ob;
         Ob = x
         ob. show();
         return 0;
}
// end of main
```

this is achieved by using parameterised constructor with only one argument

Lecturenotes.in

Converting class to basic type:-

This is done by conversion function whose general syntax is:-

```
→ Operator type()      ← to which type the class will be converted
  {
       // Body of conversion function
  }
```

no return type

Restrictions:-

→ It needs to be a member function

→ No argument

→ No return type.

Lecturenotes.in

eg:-
```
// class number begins
class member
{
      int x, y;
    public:
      member()
      {
      }
      member(int a, int b)
      {
            x = a;
            y = b;
      }
      void show()
      {
            cout << x << y;
      }
      operator int()
      {
            int temp;
            temp = x + y;
            return temp;
      }
}; // end of class member
```

Lecture Notes
Lecturenotes.in

```
// start of main

 int main()
 {
      //Declaration section

        int z;
        member ob(10,11);
      //Invoking the conversion
             function

        z = ob;
      // Display section

        ob. show();
        cout<< z; return 0;
  }// end of main
```

Output :-

```
10 11
21
```

Lecturenotes.in

7.10.10

Class To class Type:-

```
myclass ob;
myclass1 obj;

ob = obj;
```

uses uses

1) constructor
2) conversion function.

ob = obj → (source)

destination

eg:- → using constructors:-

```
// myclass begins.
class myclass
{
    int x,y;
  public:
      myclass()
      {

      }
     myclass(myclass1 obj)
     {
          x = obj.geta();
```

Lecturenotes.in

```
        y = obj.getb() + obj.getc();
  }

    void show()
     {
           cout<<"\nvalue of
                       membere:";
           cout<<x<<y;
     }
 };
  //end of myclass
 // myclass1 begins
  ~~clas~~ class myclass1
    {
          int a,b,c;
       public:
          myclass1()
          {
          }
```

LectureNotes.in

```
myclass1 (int x, int y, int z)
 {
        a = x;
        b = y;
         c = z;
  }
  int geta()
   {
     return a;
    }
     int getb()
      {
          return b;
       }
    int getc()
     {
         return c;
      }
};
```

// end of myclass1.

Lecturenotes.in

```
// start of main()

    int main()
    {
        // Declaration section
        myclass1 Obj(6,7,8);
        myclass ob;
        // conversion
            ob = Obj;
        // Display section
            ob.show();
            return 0;
    }
  // end of main
```

→ using conversion function:-

```
 // myclass1 begins.
   class myclass1
   {
         int a,bc;
```

Lecturenotes.in

```
public:
     operator myclass()
     {
          myclass obj;
          obj.getx()=a+b;
          obj.gety()=c;
          return obj;
     }
};  // end of myclass1.
// myclass begins
   class myclass
   {
        int x, y;
      public:
        myclass()
        {
        }
        ~~myc~~
        int& getx()
        {
             return x;
        }
```

Lecturenotes.in

```
mycclass (int s, int s1, int s2){a=s, b=s1,
                                   c=s2};
  int & gety()
   {
        return y;
   }

   ~~int& so~~
   void show()
    {
        cout<<"\n values";
        cout<<x<<y;
    }
}; // end of myclass.
//start of main
int main()
 {
      // Declaration section
        myclass ob;
        myclass1 obj(10,11,12);
```

Lecturenotes.in

```
// con version function
        ob = obj;
     // display section
        ob.show();
     return 0;
  } // end of main.
```

How to call const. of virtual Base?

```
 // class base begins.
 class base
 {
       protected:
           int i;
       public: base(){ i=0;}
           base(int x)
           {
                i=x;
           }
  }; // end of base
```

Lecturenotes.in

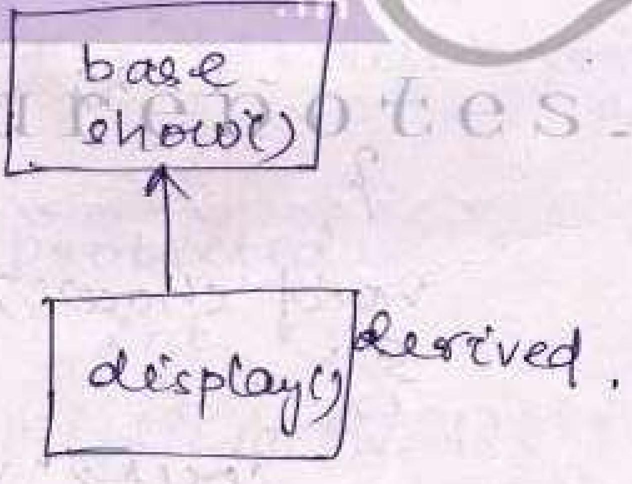
```
// derived1 begins
class derived1: public virtual
                         Base
 {
      protected:
          int g;
      public: derived1(){ g=0}.
          derived1 (int a)
           {
                j=a;
           }
  }; // end of derived1.
 //derived2 begins
 class  derived2: public virtual
                           Base
 {
          protected:
            int  k;
          public;derived2() {k=0}
            derived2(int a)
            {
                  k=a;
            }
 }; // end of derived 2
```

Lecture Notes .in
Lecturenotes.in

```
// class derived3 begins
class derived3: public derived1,
                public derived2
{
        int l;
    public:
        derived3()
        {
            l=0;
        }
        derived3(int x):
            derived1(x+2),
              derived2(x+3),
                base(x+6)
        {
            l=x;
        }
        void show()
        {
            cout<<"Values:";
            cout<<i<<j<<k<<l;
            ~~cout~~
        }
}; // end of derived3
```

Lecturenotes.in

```
// start of main

int main()
 {
        //declaration section

          deriveds ob(10);

        //display section
   Ob.
          ob.show();
message          -> Message
passing   return 0;
     }

   //end of main
```

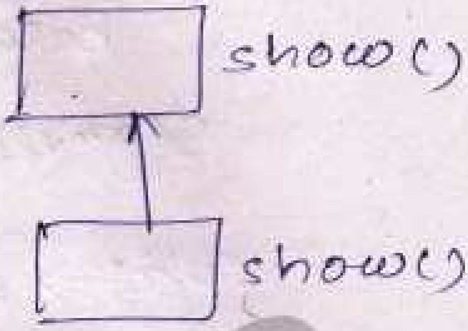


```
 base ob, *ptr;
 derived ob1, *ptr1;
   ptr= &ob;
```

Lecturenotes.in

```
ptr -> show();
ptr1 = &Ob;
ptr2 = &obl;
```

```
base ob, *ptr1;
derived obj, *ptr2;
```



```
Ob = Obl;
Ob. show();
Ob. obj.show() ×
```

Using base object, we can access base part of derived.

```
ptr1 = &ob;
ptr2 = &obj;
ptr2 -> show();
```

↳ derived class show

~~ptr1 -> show();~~

ptr2 -> base::show

↳ calls show of base

Lecturenotes.in

ptr1 → show();

↳ calls show of base

ptr1 = & obj;

↳ can be done

Base pointer can point to derived object.

ptr1 → show();

↳ Base class show invoked.

using base pointer we can access only base part of derieved class.

Because, the function call is resolved at compilation time. Compiler binds the fun^n by checking the type of pointer.

Lecturenotes.in

ptr1 → derived::show

↓

using a base pointer we cannot access the derived part.

ptr1 → show();

base ob, *ptr;

derived ob1;

ptr = &ob.

ptr → show(); → Base cla

① show called

ptr = &ob1;

ptr → show(); ②

↓

will call derived show.

Requirement

dynamic ~~run time~~ binding

→ can be done by making the base class fun^n as virtual in base class

1) ~~[illegible]~~ for making a fun^n virtual, we need to consider the following:

Lecturenotes

1) fun^n should be in both base & derived.

2) Using base pointer, we access both base and derived. fun^n

Statement ① and ② depict run-time polymorphism and use the concept of dynamic binding.

25.10.10.

Virtual function

1) Member function which is declared within a base class and redefined in a derived class.

Lecturenotes.in

eg →

```
//class base
class base.
{
        public:
        virtual void vfunc()
        {
            cout<<"\n function
                  of base";
        }
};
// class derived
class derived: public base.
{
        public:
        void v-func() // Redefi-
                          ned
        {
            cout<<"\n fun of
                   derived";
        }
};
```

Lecturenotes.in

```
// main

int main()
{
        base *p, obj;
       derived obj1;
          p = &obj;
         p -> vfun();
       ↳ calls vfun of base
                      class.
          p = &obj1;
         p -> vfun();
       ↳ calls derived class
                 fun
          return 0;
}
```

Circled statements represent run-time polymorphism because ~~diff~~ same msg is passed to diff obj.

Lecturenotes.in

Base class pointer used to call virtual functions helps achie run-time polymorphism.

Restrictions on Virtual fcn's :-

a) Virtual functions are non-static member functions. (static member functions cannot be declared as virtual).

b) The signature (name of funⁿ, no. of arguments, type) must be same in base and derived classes.

Lecturenotes.in

c) constructors can never be virtual, but destructors can be declared as virtual.

const members.

const int i=6;

then we cannot write.
i=10 later.

const int i; X not possible

const member function :-

Syntax:-

```
r-type funname()const;
```

→ cannot manipulate the data members.

→ data members become const for that particular member funⁿ

→ If we want ~~any~~ the const member funⁿ to manipulate any variable, then we declare it Mutable.

Lecturenotes.in

Syntax:-

```
mutable datatype var-name;
```

Explicit constructor:-

```
class myclass
{
        int a;
    public:
        myclass(){ };
        explicit myclass(int x)
        {
            a = x;
        }
        void display()
        {
            cout<<"\n member:"<<a;
        }
};
int main()
{
        myclass obj=45;
        obj.display();
        return 0;
}
```

Lecture Notes
Lecturenotes.in

If we don't include the key~~board~~word explicit, constructor is called for built -> class type conversion

If the keyword explicit is used before the constructor name, then it is invoked explicity.

Const object :-

```
class myclass
{
        int a;
        int b;
    public:
        myclass (){ }
        myclass (int x ;int y)
        {
                a = x;
                b = y;
        }
```

Lecturenotes.in

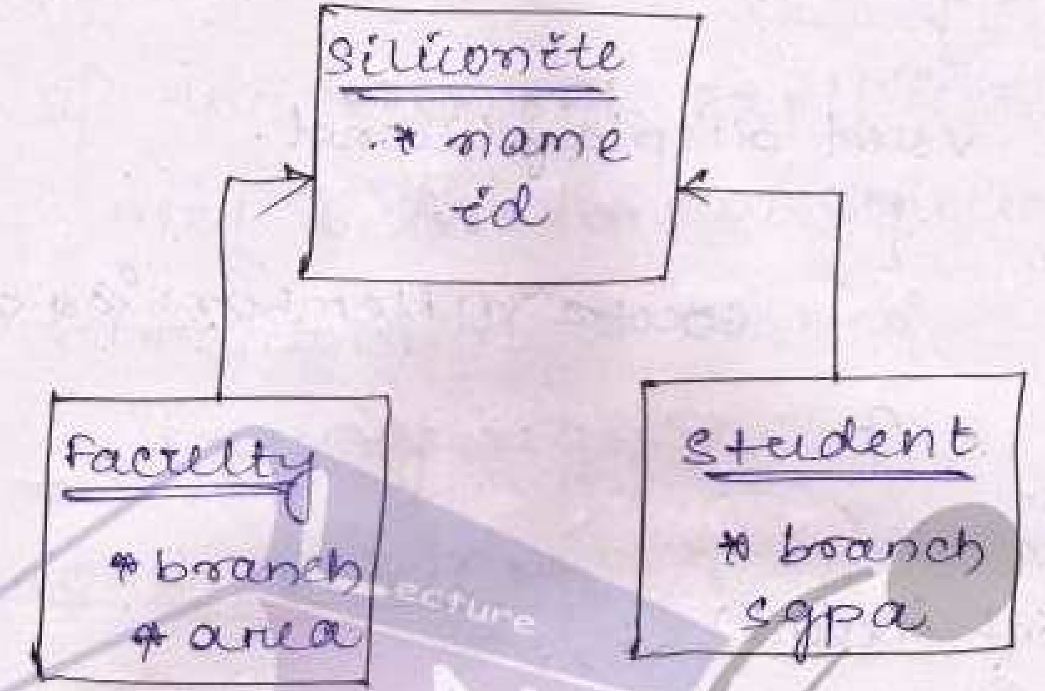
```
// void get()
  {
        a = 34;
         b = 9;
  }
  void display() const
    {
         cout<<"\n Member:"<<a<<b;
    }
};
int main()
  {
       const myclass obj(8,7);
       // obj.get();
       obj.display();
        myclass obj1(5,6);
        obj1.display();
        return 0;
  }
```

✓const object cannot access non const member func.

LectureNotes.in

→ non-const object can invoke const member fun^{ns}.

28.10.10.



```
//start of program
#include<iostream>
using namespace std;
// class siliconite begins
class siliconite
{
        protected:
                char *name;
                int id;
```

Lecturenotes.in

```
public:
    siliconite()
    {
        name= new char[20];
          id = 0;
    }
    siliconite (char* ptr, int id1)
    {
        ~~name~~ ptr= new char[20];
          strcpy(name, ptr);
            id = id1;
    }
    ~~virtual void showdetails()~~
    ~~{~~
        ~~cout<<"\n Members:";~~
        ~~cout<<"\n Name:"<<name;~~
        ~~cout<<"\n~~
      virtual void showdetails()
      {

        }
}; // end of siliconite.
```

Lecture Notes.in

```
//class faculty begins
 class faculty:public siliconite
 {
        char *branch;
        char *area;
      public:
        faculty()
        {
              branch=new
                    char[10];
              area=new
                    char[10];
        }
      faculty(char *ptr1, char *ptr
              char *ptr2, int z):
        siliconite(ptr2,z)
      {
          branch=new char[20];
          strcpy(branch,ptr1);
          area=new char[20];
          strcpy(~~branch~~ area, ptr2);
      }
```

Lecturenotes.in

```
// class student begins.
class student: public siliconite.
{
            char *branch;
             float sgpa;
         public:
             student()
            {
                branch=new char[10];
                 sgpa= 0.0;
             }
            student(char *ptr, float y,
                  char *ptr, int x):
               siliconite(ptr,x)
            {
                branch=new char[20];
                 strcpy(branch,pt);
                 sgpa=y;
             }
```

Lecturenotes.in

```
void showdetails()
 {
        cout<<"\n student's information:";
        cout<<"\n Name:"<<name;
        cout<<"\n id: "<< id;
        cout<<"\n branch:"<<branch
        cout<<"\n sgpa:"<<sgpa;
 }
};//end of student class
```

*

```
 void showdetails()
  {
      cout<<"\n faculty's information
      cout<<"\n name:"<< name;
      cout<<"\n id:"<<id;
      cout<<"\n branch:"<< branch;
       cout<<"\n area:"<< area;
  }
}; // end of faculty class
```

Lecturenotes.in

→ virtual property of a base class fun^n is inherited by derived class.

```
// start of main
 int main()
 {
       //Declaration section
         silicontite *ptr;
         student obj("EEE", "8.87";
                        "Jyotsna", 10);

            ptr= &obj;
          ptr->showdetails();
                          CS
          faculty obj1("~~EEE~~", "soft
             computing", "shalini",
                         23);

            ptr= &obj1;
           ptr-> showdetails();

          return 0;
   } // end of main
   // end of program
```

Lecturenotes.in

Pure Virtual Function:-

for giving no defination,

```
class siliconite
{
        ____
        ____
        ____
        ____
      public:
        virtual void
    showdetails() = 0;
            ↓
```

pure virtual fun^n.

<u>Syntax</u>:-

```
virtual rtype fname(arg list) = 0;
```

✓ When a class contains a pure virtual fun^n, we cannot create any object.

Lecturenotes.in

Abstract Base class:-

The (base) class whose individual object cannot be created because it contains a pure virtual function is called as a abstract (base) class. But if any class(es) is(are) derived from it then we can create their objects even if it inherits that pure virtual function.

Need:-

→ To execute hierarchy.

→ To provide interface to other classes

→ For extension

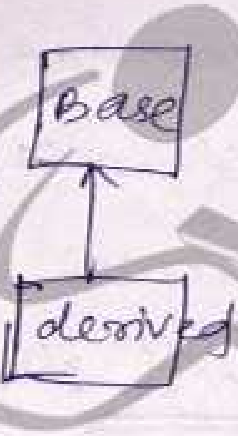
lecturenotes.in

29.10.10.

→ Constructors cannot be declared as virtual as no v-table is created for them for ~~they~~ are created by constructor itself.

→ Using base-class reference run time polymorphism can be achieved.

```
    derived obj;
     base &R=obj;obj1;

  derived &R1= obj1;
         ↳ not
              possible.
```



Lecturenotes.in

```
class Base
{
        public:
            virtual void show()
            {
                cout<<"Base class";
            }
};
```

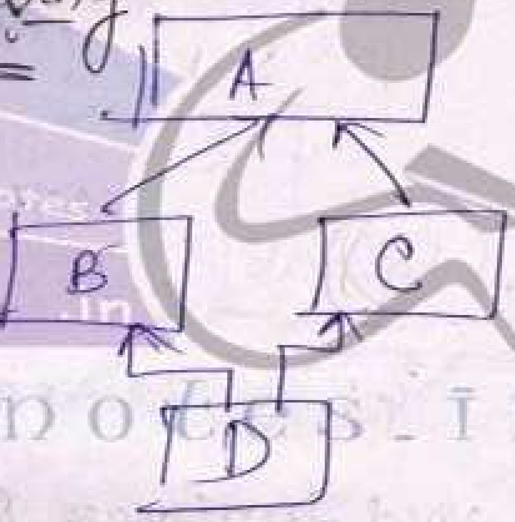
```
class derived: public base
{
      public:
          void show()
          {
               cout<<"\n derived
                              class";
          }
 };
class derived1: publicderived
{
      public:
          void show()
          {
               cout<<"\n derived1
                              class".
          }
 };
void fun(Base &R)
{
      R.show();
}
```

Lecturenotes.in

```
int main()
{
      ~~b~~ base obj;
       derived obj1;
        derived1 obj2;
        fun(obj);
         fun(obj1);
        fun(obj2);
        return 0;
}
```

1-11-10

Check the ~~[illegible]~~ following:



```
I) class A
    {

    };
  class B: public A
    {

     };
   class C: ~~p~~ virtual public A
    {

     };
    class D : public B, public C
      {
      };
```

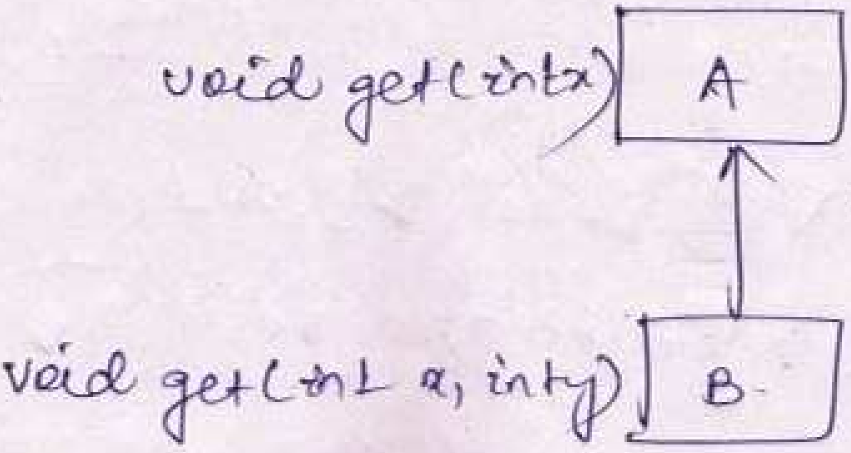
LectureNotes.in

```
int main()
{
        D obj;
        ≡
       return 0;
  }
```

II)

```
class A
{
};
class B: virtual public A
{
};
class C: public A
{
};
class D: public B, public C
{
};
int main()
{
        D obj;
         ≡
 }   return 0;
```

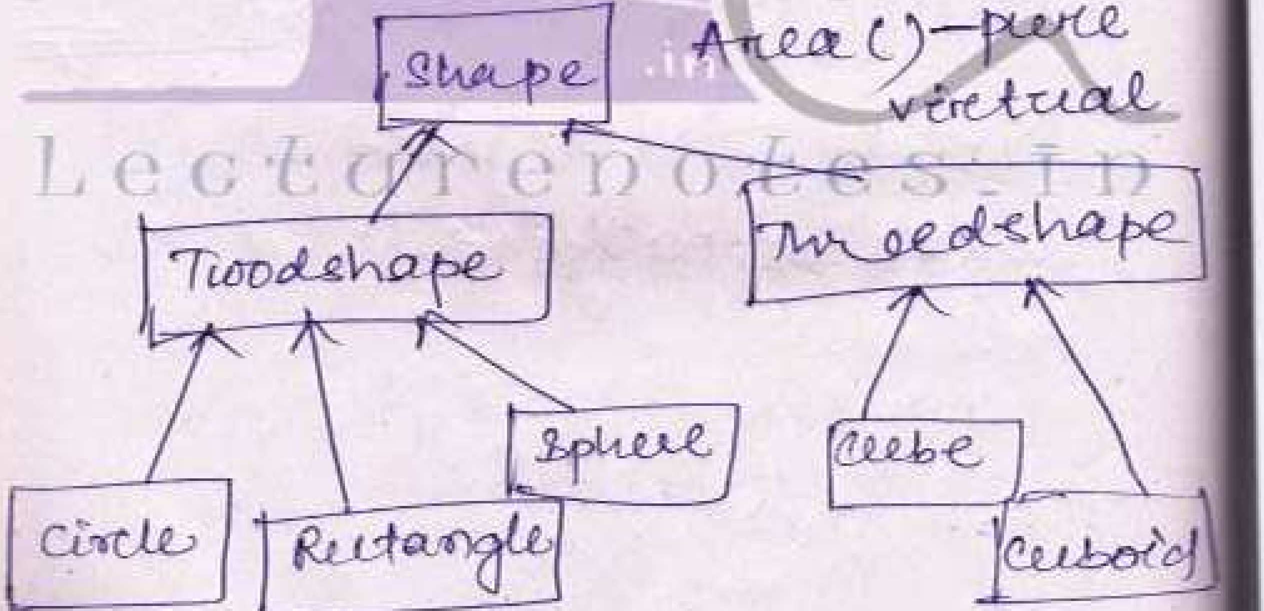
Lecturenotes.in



Check:-

```
B obj;
obj. get(5,6);

obj.@:get(5);
```

Assignment:-



Lecture Notes .in
Lecturenotes.in

Generic Programming:-

Is achieved by using templates

- Generic functions
- Generic classes

Generic Function:-

It is a function with generic data type which can operate many different data types, without writing specific code for those types

eg:- swap fun for int, char, double type.

conventionally,

```
void swap(int&, int&);
void swap(char&, char&);
void swap(double&,
              double&);
```

using the concept of generic programming, we will write a single swap function to implement swapping of all the 3 types.

Lecturenotes.in

Syntax

keyword

place holder

```
template < class Gtype >
r-type fun^n name( arg-list)
{
        Body of the fun^n
}
```

There shouldn't be any statement in bet^n

G-type is a place holder for the types that will be operated by the generic function.

eg generic swap function.

```
template < class Gtype >
void swap( Gtype &x,
                  Gtype&y)
{
        Gtype temp;
        temp = x;
          x = y;
        y = temp;
}
```

Lecturenotes.in

```
int main()
{
        int x,y;
        cout<<"\n enter integer data";
         cin>>x>>y;
        char a,b;
        cout<<"\n enter characters
                          value:";
          cin>>a>>b;
         double var,var1;
        cout<<"\n enter double value:";
          cin>>var>>var1;
        cout<<"\n calling generic fun";
          swap(x,y);
          swap(a,b);
          swap(var,var1);
        cout<<"\n swapped values\n";
         cout<<x<<"\t"<<y<<"\n";
        cout<<a<<"\t"<<b<<"\n";
        cout<<var<<"\t"<<var1<<"\n";
        return 0;
}
```

Lecturenotes.in

we can also write,

```
template<class Gtype> void
    swap(Gtype&, Gtype&)
```

```
#include <iostream>
using namespace std;
//class shape begins
 class shape
  {
        public:
             virtual void area() =0;
  };
//class 2dshape begins
 class 2dshape: public shape
  {
      public:
           virtual void area() =0;
  }
```

Lecturenotes.in

```
// class 2dshape begins
 class 2dshape : public shape
 {
      public:
          virtual void area() = 0;
 }
// class circle begins
  class circle : public 2dshape
  {
        private;
            float r;
        public;
            circle()
            {
                r=1.0;
            }

            circle(float a)
            {
                r = a;
            }
```

Lecturenotes.in

```
  void area()
  {
        cout<<"\nArea of circle:"
               <<(3.14*r*r);
   }
};
// class rectangle begins
  class rectangle: public 2dshape
  {
       private:
          float x,y;
       public:
          rectangle()
          {
               x=0.0;
               y=0.0;
          }
          rectangle(float a, float b)
          {
               x=a;
               y=b;
          }
```

Lecturenotes.in

```
    void area()
    {
        cout<<"\n Area of rectangle"
            <<(x*y);
    }
};
//class square begins
 class square: public 2dshape
 {
        private:
            float z;
        public:
            square()
            {
                z = 0.0;
            }
            square(float a)
            {
                z = a;
            }
```

Lecturenotes.in

```
    void area()
    {
        cout<<"\n Area of the
                square:"<<(y*y);
    }
  };
// class cube begins
  class cube: public 3dshape
  {
      private:
            float p;
      public:
            cube()
            {
                p=0.0;
            }
            cuble(float c)
            {
                p=c;
            }
```

Lecturenotes.in

```
void area()
  {
     cout<<"\n Area of cube:"
          <<(p*p*p);
  }
};
//class cuboid begins
 class cuboid:public 3dshape
 {
     private:
        float p,q,r;
     public:
        cuboid()
        {
            p=0.0;
            q=0.0;
            r=0.0;
        }
        cuboid(float u, float v,
                       float w)
        {
            p=u;
            q=v;  r=w;
        }
```

Lecturenotes.in

```
void area()
 {
      cout<<"\n Area of cuboid:"
           <<(p*q*r);
  }
};
// start of main
 int main()
 {
        circle obj(2);
        rectangle obj1(3, 4);
         square obj2(4);
         cube obj3(2);
         cuboid obj4(2,3,4);
        obj.area();
       obj1.area();
        obj2.area();
       obj3.area();
        obj4.area();
       return 0;
}
```

Lecture Notes
Lecturenotes.in

Template function (with two generic type)

Instantantation of generic function.

swap(x, y) → specicisation of function template/template function

swap('a','b') → Instance of generic function

→ template<class type1, class type2>

Syntax:- two generic type

```
template <class type1, class type2>
r-type f-name(type1 x, type2 y)
{
    Body
}
```

eg:-

```
template <class type1, class type2>
void display(type1 x, type2 y)
{
    cout<<x<<y;
}
```

Lecturenotes.in

```
int main()
{
        display( 10, "Hello");
        display (10, 20);
        display (9.6 , 10);
        display( 11.98, 'a');
        display("Hello", 'x');
         retturn 0;
}
```

Specific version of template function:

eg→
```
     template <class T>
     void swap( T &x, T &y)
     {
            T temp;
             temp=x;
               x = y;
               y = temp;
     }
     void swap(char *p, char *q)
     {
          char *temp;
           temp=new char[20];
```

Lecturenotes.in

```
    strcpy(temp, p);
    strcpy(p, p1);
    strcpy(p1, temp);
}
int main()
{
        int x = 10, y = 20;
        swap(x, y);
        char x1 = 'a', y1 = 'b';
        swap(x1, y1);
        char arr[] = "Hello";
        char array[] = "Student";
        swap(arr, array);
        // The special version of swap
           function will be invoked.
        return 0;
}
```

Lecturenotes.in

Overloading of function template / template function :-

A template function with different number of generic type.

```
template < class type1 > void display(
                              type1 x)
{
        cout<< x <<endl;
}
template <class type1, class type2>
void display(type1 x, type2 y)
{
      cout<< x<< y<< endl;
}
int main()
{
        display("Hello");
        display("Hi", "Students");
        display(10, 11.9);
        return 0;
}
```

Lecturenotes.in

using specific type with function templates:

eg→
```
template<class type>
void display(type x, int y)
{
      cout<<x<<y<<endl;
}
int main()
{
        display(10.9, 5);   always integer type
        display("Hello", 6);
        display(10, 20);
        display(10.0, 20);
        return 0;
}
```

Lecturenotes.in

Generic class:

The syntax of defining a generic class.

```
template<class type>
class classname
{
```

```
// Body of class
};
```

eg→

```
template <class type>
class myclass
{
      private:
            type x;
            type y;
      public:
            myclass()
            {
            }
            myclass(type a, type b)
            {
                  x = a;
                  y = b;
            }
            void display();
};
```

Lecturenotes.in

* To define the member-function of generic class outside the class */

Syntax :-

```
template<class type> r-type classname<type>:: f-name(arg list)
{
        body of the function
} */

// defination of display
template<class type> void myclass<
        type>::display()
{
    cout<<"\n member of the
                class\n";
    cout<<x<<"\t"<<y;
} // end of function
// To implement generic class
int main()
{
    myclass<int> obj1(5,6);
    myclass<char>obj2('a','b');
```

Lecturenotes.in

```
myclass <double> obj3(2.6, 7.7)
    obj1. display();
    obj2. display();
    obj3. display();
    return 0;
}
```

→ The general syntax for creating the instance of a generic class is

```
classname <specific type> object-
                          name;
```

<u>class with two generic type:-</u>

<u>Syntax:-</u>

```
template <class type1, class type2>
class class_name
{
        Body of class
};
```

Lecturenotes.in

// To define the object of a generic class with multiple (two) generic type.

Syntax:

```
classname <1st specific type,
        2nd specific type> object_name;
```

eg:

```
template<class type1, class type2>
             class myclass
   {
      Body of class

   }
   myclass<int, char> obj;
```

class with non-generic type:

Lecturenotes.in

Syntax:

```
template< class type1, int size>
 class class_name
  {
        type1 x;
         type2 y;
  };
```

Eg:-

```
template<class type1, int size>
  class array1
  {
       type1 A[size];
          =
          —
  };

  Array<int, 10> obj;
```

Q) create a generic stack class with required member functions (push, pop) and implement the class for 1) integer 2) char 3) double types of data.

12.11.10.

```
// class begins
template<class type, int size>
 class stack
 {
       type *stack;
         int tos;
     public:
        stack()
        {
              tos=-1;
```

LectureNotes.in

```
    stck = new type[size];

  }
  void push(type);
  type pop();
}; // class ends.
// defination of push()
 template<class type>void stack<type>
                ::push(type x)
 {
        if(tos== size)
           cout<<"\n stack is full";
           return;
        else
          { tos++;
               stck[tos]=x;

          }
  }
// defination of pop
 template<class type>type stack<type>
                ::pop()
 {
         type y;
```

Lecturenotes.in

```
if ( tos ==-1)
{  cout<<"\n stack is empty";
     return;
  }
else
 {
      y= stack[tos];
       tos--; return y;
   }
}
// implement the class stack.
 int main()
{
      stack<int, ~~size~~ 20> obj;
       /* An object of class stack
         is created with a intege
          stack */
      ~~Obj. push(1)~~;
      ~~Obj.~~ for( int i=0; i<20; i+
                  Obj. push[i];
```

Lecturenotes.in

```
for (int j=0; j<20; j ~~++~~)
      cout<<"\n"<<obj.pop();
  return 0;
}

template<class type=char,
                int size=20>~~&~~

class stack
 {


 };
 stack<int, 30>~~&~~ obj;
   // obj will be created for
        integer stack of size 30.
 stack<double>~~20~~ obj1;
   // obj1 will be created for
        double stack of size 20.
 stack<> obj2;
   //obj2 will be created for
        char stack of size 20.
```

Lecturenotes.in

Exception Handling:-

Exception:- An error which abnormally terminates the program.

eg:-
```
#include<iostream>
using namespace std;
int main()
{
        int x, y, z;
        cout<<"\n enter the value of
                x, and y: ";
        cin>>x>>y;
        z = (x/y);
        cout<<z;
        return0;
}
```

If x =10, y =0.
Program ~~will~~ may terminate at
z= (x/y);

1) Division by zero
2) Memory not allocated
(requested memory)

Lecturenotes.in

5) Using a memory which is not allocated to program

C++ handles these kind of exceptions by using three keywords.

1) try → block statement

2) throw → simple statement i.e; used to throw the exception from try block to catch block

3) catch → block statement

```
int main()
{
        try{.
                if(y==0)
                    throw y;
                else
                    z=(x/y);
        }
        catch(int a)
        {        if(a==0)
                 { cout<<"\n Reenter
                                    y";
        }        }cin>>y;
```

Lecturenotes.in

15.11.10.

Exception Handling:-

Exception:- An abnormal cond^n in a program which cause the program to terminate abruptly.

→ The program segment which is expected for generating exception is put in the try block. If the exception occurs then it is thrown to catch block by using keyword ~~catch~~ throw.

Syntax:-

```
throw exception;
```

where exception can be of any type (data type).

The exception thrown by try is catched by the catch block

Syntax:
```
catch(arg)
{

}
```

Lecturenotes.in

Syntax :-

```
try
{

    ______
    ______
    ______
    ______

    throw exception; // throw
                        part
    ______
    ______
    ______
}
catch(arglist)
{


}
```

Lecturenotes.in

→ A try block can be associated with multiple catch blocks for an exception which catch block will execute, that is decided by the type of exception thrown

```
try
{
      ≡
      throw exception;
      ≡
  }
  catch (arg)
   {

    }
   catch(arg)
    {

      }
    catch (arg)
     {

       }
```

Lecturenotes.in