OOPs

18-1-18

Structure in C

In C when a structure is defined, it contain only data members or attributes defining the structure. All the attributes may belong to different datatypes. To use the structure user can define structure variables. The structure variables access the member of structure using dot operator.

The syntax is :-

```
struct variablename.membername;
```

Using this seperate memory is allocated to each structure variable.

Example :-

```
struct student
{   char name[10];
    int rollno;
    float marks[5];
    int phone no;
} s1, s2, s3, s4, s5;

struct s1.name;
```

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

Limitations of Structure in C :-
In C language, user cannot define functions in a structure. Whereas this can be done in C++.

→ In C++.
#include <iostream.h>
Cin >> → extraction operator
Cout << → insertion operator

Example :-
```
Cout << "Enter value of a and b";
Cin >> a >> b;
```

- Function :-
Declaration :- int Sum(int, int);
Definition :-

Functions are not allowed in the structure in C language.

19.1.18

Structure in C++ :

```
Struct student
{
    Char name[10];
    int rollno;
    void input()
    {
        Cout << "Enter name of student";
        Cin >> name;
        Cout "Enter name rollno of student";
        Cin >> rollno;
    }
};

void main()
{   struct student S1, S2;
    S1.input();
    S2.input();
    getch();
}
```

4-1-18

In C++ User can define function within the structure. These functions can be called multiple number of times using different structure variable.

<iostream> → class.
Cout → Console output object
Cin → Console input object.
>> → Shift right (/2)
<< → Shift left (*2).

Classes are the user defined datatype which combines datamembers and member functions in a single unit. Datamembers defines the attributes of the entity represented by class and member function defines the operation that can be performed on those attributes. There are two access specifier → Private and Public → that can be used in a class. These access specifier defines the ~~that~~ right to access the members of the class.

Object is the instance of the class. One class can have multiple objects. using the objects of the class, user can access the public members of the classes with the help of (.) operator.

→ Area of Rectange :-

```
class rectangle
{
  private : int l, b;
  public : void input()
      { cout << "Enter l and b";
        cin >> l >> b;
      }
      void area()
      { int ar;
        ar = l * b;
        cout << ar;
      }
}
```

```
void main()
{
  rectangle r1;
  r1.input();
  r1.area();
  getch();
}
```

25-1-18

* Features of Object Oriented languages:-

1) Data hiding and encapsulation :- OOPs languages allows the user to combines the ~~data~~ data members and member function in a single unit known as classes. This property is known as ~~data~~ encapsulation. User can hide the important information defined in the class using private access specifier. This property is known as data hiding. The data which is written under private access-specifier cannot be accessed directly by the object of the class. They require public member function to access the private ~~nor~~ data member of the classes.

2) Data Abstraction :-
In Object Oriented languages user can abstract the detail required at lower level from the higher level as per the requirement. This property is known as data abstraction. Data Abstraction can also be defined as showing the important data and hiding the unimportant data.

3) Friend Function :-
It is an external function which does not belong to same classes and can accesses private data members of the ~~as~~ class. Friend function represent the alternate path to data hiding
keyword :- friend.

29-1-18

4) Inheritance :-
5) Operator Overloading
6) Polymorphism. < static / Dynamic
7) Exception handling.

```
#include <conio.h>
#include <iostream.h>
class rectangle
{
private: int l, b;
public: void input()
        {
          cout << "Enter l and b";
          cin >> l >> b;
        }
        void area()
        {
          int ar;
          ar = l * b;
          cout << ar;
        }

        void perimeter()
        { int p;
          p = 2 * (l + b);
          cout << p;
        }
}; void main()
{ rectangle k;
  k.input(); k.area();
  k.perimeter(); getch();
}
```

Member function of a class can be defined outside the class using scope resolution operation

Scope resolution operator → (::)

31-1-18

* Definition of member function of class outside the class :-
using scope resolution operator (::)

Syntax :-

```
return type classname :: function name()
{
    Body of the function
}
```

Example :-

```
class rectangle
{
    int l, b;
public: void area(){
    { cout << l*b; }
    void input()
    { cout "Enter l & b"; cin >> l >> b; }
    void perimeter();
};
```

Private is default specifier.

```
void rectangle :: perimeter()
{
    cout << 2*(l+b);
}
```

1) There is no need to pass parameters to the member function of the class.
2) Any no. of objects can be created corresponding to one class and each object can be initialised with a different set of values for private datamembers using input member function of the class.
3) The private data members are not initialised in the class at the time of declaration. By doing so each object will have some value of private datamember

* Constructor :-

1) A Constructor is a special public member function of a classes which can initialised the private datamembers of a class.

2) The name of Constructor is same as the name name of class.
3) The invoking of constructor is simple. It is called automatically when an object is created.
4) There are three types of Constructor:
   - Default
   - Parameterised
   - Copy
5) In a class user can define more than one constructor at same time. This property is known as constructor overloading

1-8-18

Example :-

```
class rectangle
{ private : int l, b;
  public : rectangle()
           {
               l=5; b=2;
           }
           rectangle (int x, int y)
           { l=x;
             b=y;
           }
```

Default Constructor

Parameterised Constructor

& is the command to the compiler to access the object through address.

```
void area()
  { cout << l*b; }
};

Void main()
  { rectangle t;          → (Calling default Constructor)
    rectangle t1(3,2)     → (Calling parameterised constructor)
      t. area();
      t1. area();
    getch();
  }
```

→ Copy Constraints :-

```
classes rectangle
{ private : int l, b;
  Public : rectangle (rectangle &R)
           { l = R.l;
             b = R.b;
           }
           rectangle (int x, int y)
           { l = x; b = y; }
           void area()
           { Out << l*b; }
};
```

```
Void main()
  { rectangle t1 (6, 5);
    rectangle t2 (t1);      → Calling Copy Constructor
    t1. area;
  }
```

t1 → source object
t2 → destination object

* Friend Function :-

1) A friend function is an external function which can access the private datamembers of a class
2) It does not belong to class i.e it is not the member function of class
3) Friendship is granted by the class to which that function is friend
4) A friend function is declared as a friend to a class using keyword : Friend
5) A friend function is also define outside the class
6) A friend function is called without the object of the class.
7) The object of the class to which function is friend is passed as parameter or argument.

2-2-16

Friend Function

```
class rectangle
{
private: int l, b;
public : rectangle (int x, int y)
        { l=x;
          b=y;
        }
    friend int area (rectangle &r);
};

int area (rectangle &r)
{
    int ar;
    ar = r.l * r.b;
    return ar;
}

void main ()
{
   rectangle r(4,5);
cout<<area(r);
  getch();
}
```

```
class square ; → [fre declaration]
class rectangle
{   private : int l,b;
    public : rectangle (int x, int y)
            { l=x; b=y; }
    friend void area (rectangle &x, square &s);
};
class square
{   private : int a; // declaration
    public : square (int a) { a=a; }
    friend void area (rectangle &r, square &s);
};
    void area (rectangle &x, square &s)
    {   int arr, ars;
        arr = x.l * x.b;
        ars = s.a * s.a;
        cout<<arr<<ars;
    }
void main()
{   rectangle r(2,3);
    square s(4);
    area(r,s);
    getch();
}
```

* Difference Between friend function and member function.

1) Friend function is defined Outside the class whereas member function is defined in the class or outside using (::).
2) Friend function ~~belongs~~ can be related to one or more class but member function only belongs to one class.
3) There is no need to pass the parameters using object to the member function but for friend function there is a need to ~~must~~ pass the parameters.
4) Access specifier is applicable to ~~friend~~ member function but not on friend function.
5) Member function is called using the object whereas friend function is called by passing the object.

5-2-18

* Friend Class :-

When two or more functions are friend to same classes then we can collect both function together in a single class name as friend class. If class A ~~A~~ is grants friendship to class B then all the members of class B can access the private members of class A.

Characteristic of friend class :-

1) If Class A is a friend of class B, it does not imply that B is a friend of A which means that C++ friendship is not reciprocating.
2) C++ friendship is not transmitted.
3) A class is declared as friend of another class by using keyword friend.
4) One class can be friend to more than one class.

Ques :- Create a class rectangle having private datamembers h, l and b and public member function ~~SA~~ perimeter which calculate the perimeter of rectangle. Create a friend class paint which calculate the cost of painting the rectangle, having rate as private datamember and cost public

rectangle: cuboid
S.A=perimeter

```
class cuboid
{
    int l,b,h;
public: cuboid(int x, int y, int z)
    { l=x;
      b=y;
      h=z;
    }
    int area()
    { int sr;
      sr= 2*[lb+bh+lh];
      return sr;
    }
    friend class paint;
};

class paint
{ int r;
  public: paint(int a)
          { r=a;
          }
    void cost(cuboid &c)
    { int tc;
      tc= r * c.area();
      cout<<tc;
    }
```

```
void main()
{
    cuboid c(3,1,1);
    paint p(5);
    p.cost(c);
    getch();
}
```

7-2-18

Static Data Members :-

Static data members are those members whose common value is shared by all the objects of a classes. In general, in case of non-static data members whenever an object is created a memory space corresponding to the private data member is allocated to the object. Each object can store its own value for the data members. In some special cases when each object has same value for a particular datamember, that datamember is declared as static. A static datamember has same value for all the objects of the class. This

saves the memory space.

→ Characteristics of static data members:-

1) A data member is declared as static by using keyword 'static' at the time of declaration.
2) A static datamember has same value for all the object.
3) A static data member is not initialised with the help of the static constructor.
4) A static data member is initialised outside the class using the scope resolution operator.
5) Value assigned to a static datamember can be changed.
6) A static datamember is initialised to 0 by default.
7) Static datamembers can be declared under any access specifier.

→ Constructor are the special members that are used to initialise non-static data members.

Non-static member function ←

```
class cuboid
{
private: static int l;
         int b, h;
public : cuboid( int x, int y)
         {
           b=x; h=y;
         }
         void volume()
         {
           int v; v=l*b*h;
           cout << "Volume is" << v;
         }
};
int Cuboid :: l=5;
void main()
{
  Cuboid c(3,2);
  c.volume();
  getch();
}
```

9-8-18

Static Member function :-

1) A static member function is a member function which perform operation only on the static data members of the class.
2) Static Member function do not need object for the calling.
3) Syntax for static member function is
   classname :: static mfname();
4) A member function is declared as static by using keyword static.

Ques: Program →
Cuboid → static (l, b) Non-static (h.)
baseareal() ? Volume()

```
class cuboid
{
    int h;
    static int l, b;
    public : cuboid(int x)
             {h=x;}
    void vol()
    {   int vl;
        vl = l*b*h; cout << vl;
    }
static void baseareal()
    {   int ar;
        ar = l*b;
        cout << ar;
    }
};
int cuboid :: l=2;
int cuboid :: b=6;
void main()
{
    cuboid c1(6);
    c1.vol();
cuboid :: baseareal();
    getch();
}
```

~cuboid(); → Destructing.

12-2-18

* Pointer to class and Pointer to object of class:

To access the objects of a class at run time user can create :-

1) Pointer to object of a class
2) Pointer to a class

→ Pointer to Object of a class: → It points to a particular object of a class.

Syntax :-

```
rectangle r;
rectangle *ptr;
ptr = &r;
(*ptr).area();
or
ptr -> area();
```

indirection operator

→ Pointer to Class → It points to the classes. It is different from pointer to Object as it can point to an array of objects. (multiple objects).

Syntax :-

```
rectangle r;
rectangle *ptr;
ptr = &rectangle(r);
Or
ptr = &(rectangle) r;
ptr -> area();
```

* Dynamic Allocation and deallocation of objects:- There are two operators: one is new operator and other is delete operator. New operator can create the objects and return the address of the object created. Delete operator deletes the object already created and free the memory space.

→ Differentiate between static and dynamic allocation of memory.

~ → tilde (Destructor)

15-2-18

* Creation of Object Dynamically :-

```
rectangle * ptr;
ptr = new rectangle;
ptr → Area();
delete ptr;        [for Deleting the memory]
```

New operator can also create Array of objects.

```
rectangle *ptr;
[Constructing] ptr = new rectangle[10];
[Destructing] delete [] ptr;
```

→ Pointer to Class	Pointer to object
1) It point to array of object	1) It point to single object.
2) rectangle * ptr; ptr = &(rectangle)x;	2) rectangle x; rectangle * ptr; ptr = &x;
3) ptr ++ points to next Object in Array.	3) ptr ++ is not allowed.
4) ptr = new rectangle[10];	4) ptr = new rectangle;
5) delete [] ptr;	5) delete ptr;

16-2-18

* 'This' Pointer :-

1) 'This' pointer is a default pointer which is created by the compiler to store the address of current object in use. User cannot change the value of 'this' pointer.
2) User can use 'this' pointer to represent the object for calling the member function of the class.

Eg :- rectangle x;
x.area;
or this → Area();
or (*this).area();
Cout << this;
or Cout << &x;

19-8-18

Operator Overloading

The re-definition of the functionality of operator for user defined datatype is known as operator overloading. In the compiler the operators are defined for primary datatypes. for example :- + operator can add two integers or two real numbers. By using operator overloading, user can redefine + operator to add two objects of the classes or (two different classes).

→ In order to define the functionality of an operator, an overloading function is required to be defined by the user.

→ The Name of overloading function should be "operator <sign of operator>"

→ The overloading function can be define as the member function of the class or friend function of the classes.

→ In case when the operator is overloaded to perform operation on objects of same class, define the overloading function as member function of the class.

→ In case the operator is overloaded to perform operation on objects of two different classes, define the overloading function as friend function.

21-8-18

```
class Complex
{
  private : int r, imag;
  Public : Complex ( int x, int y)
           { r=x;
             imag = y;
           }
  Complex operator +( Complex &C)
    { r = r+C.r;
      imag= imag + C.imag;
    }
    return (*this);
}
```

```
void display()
{
  cout << " Complex no. is " << r << "+i" <<
          img;
}
};
void main()
{ Complex C1(3,2), C2(4,5), C3(0,0);
    C3 = C1+C2;
    C3.display();
    getch();
}
```

Ques:- Program to multiply a Complex number.

```
class complex
{
  int r, img;
public: complex(int x, int y)
        { r=x;
          img=y;
        }
```

```
Complex operator *(complex &c)
{  r = r*(c.r) - (img*(c.img));
   img = r*(c.img) + (img*(c.r));
   return (*this);
}
void display()
{
  cout << " Complex no. is " << r << "+i" << img;
}
};
void main()
{ Complex C1(4,6), C2(3,4), C3(0,0);
    C3 = C1+C2;
    C3.display();
    getch();
}
```

22-9-18

* Restriction on Operator overloading :-

1) Operator overloading cannot change the arity of operator.

2) There are 5 operators that cannot be overloaded. . , : : , sizeof , ?: , *

Console → computer screen

28-2-18

Overloading of >> and << operator :-
>> → extraction
<< → insertion.

Overloading insertion and extraction operator means that the operators are redefined for user-defined datatype. Extraction operator is used with cin (object of istream class) to input the data from the console) and insertion operator is used with cout (object of ostream class to display the data on the screen). These two operators can be overloaded to input or output user defined datatype (object of the class).
The overloading function for both the operator is defined as friend function as there are two different object to be used in the statement.

```
# include <iostream.h>
class complex
{
private : int r, img;
public : complex(int a, int b)
         { r=a;
           img=b;
         }
friend ostream & operator << (ostream &o,
                              complex &c);
friend istream & operator >> (istream &i, complex &c);
};
```

Containership.

14-3-18

Unit :-3

Inheritance

Use :- Reusable.

Inheritence "is a relationship"

There is no private access specifier

Containership → "has a relationship"

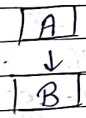
Inheritence →	Public	Protected	Private
Base M.F ↓			
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	N.I*	N.I*	*N.I.

15-3-18

Inheritence is the property of C++ which allowes the users to inherit a new classes from a pre-existing classes. The derived class can use the features of base class as well as it can has its own features. The most important advantage of inheritence is reusability of base class. Depending upon the number of base class and derived class inheritence can be of 5 types

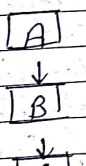
Syntax :-

1) Single :-




```
Class A
{
};
Class B : public A
{
};
```

2) Multiple :- Multi level

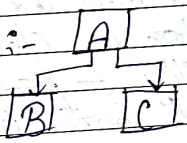



```
Class C : Public B
{
};
```

3) Multiple :-

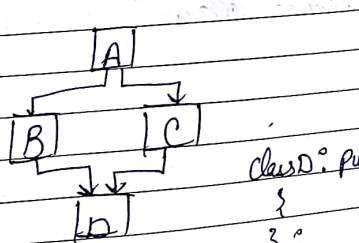

```
Class A
{
};
Class B
{
};
class C : public A, public B
{
};
```

4) Hierarchal :-




```
class B : public A { };
class C : public A { };
```

5) Hybrid :- Multipath




```
ClassD : public B, public C
{
};
```

1) Using inheritence, the members of base class can be made visible in derive Class.
2) The visibility mode of base class members in derived class depend on the type of inheritence (public, private or protected)
3) The datamembers declared as private in base class are not visible in derived class which means that private datamembers are not inherited. Thus a new access specifier protected is used in base classes.
4) In the main program, the object of derive class can accesses the members of both base class as well as derive classes.
5) The constructor of derived class has the responsibility to initialise the datamembers of both base class and derive class

Lab

15-3-18

Overloading of >> and << operator

```
class distance
{ private: int m, cm;
  public : distance()
           { m=0, cm=0;
           }
friend istream & operator >>( istream &i, distance &d)
friend ostream & operator <<(ostream &o, distance &d)
};

istream & operator >>( istream & i, distance &d)
{
    cout << "Enter distance in m &cm";
      i >> d.m;
      i >> d.cm;
    return i;
}
ostream & operator <<(ostream &o, distance &d)
{
      o<< d.m;
      o<< d.cm;
    return o;
}
```

```
Void main()
{
    distance d;
    cin >>d;
    cout << d;
    getch();
}
```

19-3-18

Program →

Inheritence

```
class base
{
Protected: int x;
Public: base(int a)
        {x=a;}
    int getx()
    {
        return x;
    }
};
```

```
class derived : public base
{
Protected: int y;
Public: derived(int b, int a) : base(a)
        {y=b;}
    int product()
    {
        int p;
        p = x*y;
        return p;
    }
};
```

```
void main()
{
    derived d(3,4);
    d.product();
}
```

```
int product
{
    int p;
    p = y * getx();
    return p;
}
```

In inheritence when there are member function having same name and same prototype But one existing in base class and other in derived class, In such case the derived class object will always call the member function of derived class only which means that definition of derived class function overrides the function definition of base class function. This is known as overriding function.

21-3-18

Class:

Father → agef, father(), display()
Son → ages, son()
daughter → aged, daughter()

```
class father
{ protected: int agef;
  public: father(int x)
        { agef = x;
          cout << "This is father class constructor";
        }
  void display()
  { cout << "Age of father" << agef;
  } ~father(){cout << " father distructor";
};
class son : public father
{
  protected: int ages;
  public: son(int x, int y): father(x)
        { ages = y;
          cout << "This is son class constructor";
        }
  void display()
  { cout << "Age of son " << ages;
  } ~son(){ cout << "son distructor";};
};
```

```
class daughter : public father
{
  protected: int aged;
  public: daughter(int x, int y): father(x)
        { aged = y;
          cout << "This is daughter class";
        }
  void display()
  { cout << "Age of daughter " << aged;
  }
};

void main()
{
  son s(45, 20);
  daughter d(47, 19);
  s.display();
  s.father::display();
  d.display();
  d.father::display();
  getch();
}
```

Output :-
This is father class constructor
This is son class constructor
This is father class constructor
This is daughter class
Age of son 20
Age of father 45
Age of daughter 19
Age of father 47
Son distructor
father distructor

22-3-18

* Containership (has a relationship)
[A has a B
A contain objects of B]

1) C++ allows the user to contain object of one class as the member of another class such a relationship is known as containership or has a relationship. The class which contains the object of another class is known as Container class and the class whose object is contained is known as Contained class.

2) In containership, the constructor of container classes passes arguments to the constructor of contained class. The Members of container class will be classes with the object of container classes and the member of contained class will be called with the object of contained classes

Example :- xxy.

```
class B
{ private : int x;
  public : B(int a)
    { x=a;
    }
  int getx()
  { return x;
  }
};
```

```
class A
{ private: int y;
  public: A(int a, int b): objb(a)
          { y=b; }
    B objb;
       int product()
       { int p;
         p = y * objb.getx();
         return p;
       }
};
void main()
{
   A obja(3,4);
cout << obja.product();
   getch();
}
```

3-3-16

* **Virtual Base class:**

In multipath inheritence, the variable declared in uppermost base (indirect base class) is inherited in the derived classes (lowest level) through ∄ Multiple paths to overcome the ambiguity of multiple instances of variable defined in indirect base class through multiple path, the indirect base class is declared as virtual base class. ~~using~~ declaring A as virtual will allow the single instance of variable in A to be inherited in lowermost derived class. This is done to maintain data integrity.

* **Run-time Polymorphism and Virtual functions:**

Ques:- Base class :-

```
class base
{ protected: int x;
  public: base()
          { x=0;
            cout << "This is base class Constructor";
          }
          base(int a) : x(a)
          { cout << "Base class Constructor"; }
  ~base()
  { cout << "Base class destructor"; }
  void display()
  { cout << "Value of base" << x;
  }
};
```

```
class derived
{ protected: int y;
  public: derived()
          { y=0; }
          derived(int a, int b) : base(a)
          { y=b;
            cout<<"Derive class constructor";
          }
  ~derived()
          {cout<<"Derive class destructor";}
  void display()
  { cout<<"Value of derived"<<y;}
};
void main()
{ derived d(3,5);
  d.display();
  d.base::display();
  base *bptr;
  bptr = new base;
  bptr -> display();
  derived d(3,5);
  bptr = &d;
  bptr -> display()
  delete bptr;
  getch();
}
```

Output :-

```
Base class constructor
Derive class constructor
Value of derived 5
Value of base 3
This is base class constructor
Value of base 0
Base class constructor
Derive class constructor
Value of base 3
Value of derived 5          (if Virtual)
Base class destructor
Derive class destructor     (if Virtual)
```

21-8-18

Pure Virtual Function :-

A Pure Virtual Function is a function which has no definition. A Virtual function is declared as pure Virtual function by equating to 0

Eg :- Virtual void display()=0;

{

}

A class which contain one or more pure virtual function is known as Abstract class. An abstract class is a class whose object is not created. It is only used for the purpose of data abstraction which means abstracting the data from base class to derived class and using it through the functions of derived class.

Virtual Destructor:-

In inheritence the base class pointer can store the address of object of base class as well as derived class. When the destructor of base class is called using delete operator it deletes the pointer or object of base class. In order to call the destructor of derived class the destructor of base class is declared as virtual. By doing so the delete operator calls the destructor of of derived class as well as base class. This concept is known as stack unwinding. The order of calling of destructor is reverse of the order of calling of constructor in inheritence.

28-5-19

Unit :- 4

* I/O Streams and files:-

There are two ways to transfer the data to program and from program i.e. one is iostream and other is filestream. In iostream using iostream header file and cin and cout user can take input from console and display the result at console. In files stream user can read the data from files and store the result in files.

File stream:-

There are two types of files which are used for data transfer. One is random access file (pointer used) And other is sequential access file

→ Input stream Declaration:

ifstream infile "cy2";

↓ header file or class — ↓ object — ↓ filename.

get()
getline()
read()

9-4-18

File ostreams are used to handle the data transfer between files and program. ifstream handle input file and ofstream handle output file. The program can read the data from input file using functions defined in ifstream class (get, getline()). The program can write the data in output files using functions defined ofstream classes (put())

Each file is associated with the name and extension. Name is given to the file user and extension is given to the system.

* File related Operations :-

1) Creating a file and assigning name to it
2) Opening a file
3) Reading or writing, Appending, Modifying, Deleting
4) Deleting errors if any.
5) Closing a file.

* Input stream declaration :-

→ It is used to associate a user created file with the library classes. To use the file as input file it is required to associate with the ifstream class using an object (created by user).

* Output stream declaration :-

To use the file as output file it is required to associate with ofstream class using an object. This is known as output stream declaration.

→ Input stream declaration associates the file which already exists (created) whereas output stream declaration creates the file if it doesn't exist.

* File Opening mode :-

a) Input (in)
b) Output (out)
c) Append (app)
d) Binary (bin)
e) no create
f) no replace
g) trunc
h) Ate

ios

Obj1.open(ios::app)

4-4-18

* Reading from a file:-
1) Using object of ifstream classes
2) Using getline() function
3) Using get() function

* Writing in a file
1) Using object of ofstream class
2) Using put();

Writing in a file means storing data in a file. This can be done by using the object of ofstream class linked to a file.

a) By using put().

Before writing in a file the file should be declared as output file using output stream declaration.

ofstream outfile("My file");
(outfile: object name; "My file": Name of file)

Once the file is associate with ofstream class user can write in the file by using the object declared in output stream declaration and insertion operator. Outfile << "This is answer";
Outfile << ans;

Using put() → The syntax of put() is:
Outfile.put('E');
Outfile.put(ch); ch=variable.

Put() display 1 character at a time.

* Reading from a file:-
ifstream Infile("Myfile");
Infile >> a;
infile.get(ch);
infile.getline(str, 10);

* File Pointers:-
When a file a modified it is imp. to mention the point or location where that modification is being done. A file pointer points to a particular position in a file. It does not store the address of the location

reference position :- Beginning → Beg
Current → Cur
END → end

either it stores the off-set (no. of bytes) from a reference position, that reference position can be beigning of the file or end of the file.

05-4-18

* File Pointers :-

predefined → Input Pointer (get pointer)
→ Output Pointer (put pointer)

Four functions to access pointers :-
1) seekp (offset, reference position)
2) seekg (offset, reference position)
3) tellp ()
4) tellg ()

→ seekg assign a new value to get pointer (reference position + offset)
→ tellg returns the get pointer value.
→ seekp assign a new value to put pointer.
→ tellp returns the put pointer value

```
ofstream outfile ("ab.");
outfile.open();
outfile << "let us learn C++";
outfile.close();
```

9-4-18

* Stream Manipulators :-

I/O
- formatted I/O (ostream manipulators)
  - Non-parameterised (defined in iostream)
  - Parameterised (defined in iomanip)
- Unformatted I/O
  - Using cin
  - Using cout

- endl → end of line
- hex → hexadecimal base conversion
- dec → decimal base
- oct → octal base

- setfill (char c) → set fill character
- setw (int x) → set width to x
- set precision (int x) → set precision of float no. to x
- setiosflag (long f) → set format flag denoted by f
- resetiosflag (long f) → reset format flag denoted by F.

```
cout << a << endl << b;
cout << setw(8) << "son";
cout << set w(15) << "none";
```

11-4-18

* Generic Programming Using Templates.

STL → Standard Template Library.

C++ allows the user to define a generic program using templates. To support the use of templates STL library is used. Using templates a program is designed by creating a place holder for data type. The data type is given a particular value at the time of usage. It is also known as template programming. There are two types of template :-

1) Function templates
2) Classes templates

Syntax :- for declaration.

```
template < class T>
```

keyword keyword generic datatype.

→ WAP to add two integers using function

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

void main()
{
    int add(int, int);
    int a, b, c;
    cout << "Enter a & b";
    cin >> a >> b;
    c = add(a, b);
    cout << "Sum = " << c;
    getch();
}
int add(int x, int y)
{   int z;
    z = x + y;
    return z;
}
```

```
#include <iostream.h>
#include <conio.h>
template <class T>
void main()
{
    T add(T, T);
    T a; T b; T c;
    cout << "Enter a, b";
    cin >> a >> b
    c = <int>add(a,b);
    cout << "sum =" << c;
    getch();
}
T add(T x, T y)
{   T z;
    z = x + y;
    return z;
}
```

→ WAP to class using template.

```
template <class T>;
class rectangle
{ T a; T b;
  public : rectangle (Tx, Ty)
           { a=x;
             b=y;
           }
    void area()
    { T ar;
      ar = a*b;
      cout << "Area=" << ar;
    }
};

void main()
{
  rectangle <float> r(3.5, 4.8);
    r. area();
  getch();
}
```