UNIT-ONE

Concepts of OOPs

Programming paradigms →

By paradigm, one means a way of thinking or doing things.

Paradigm means organizing principle of a program. It is an approach to programming.

(1) **Procedural programming →**

A program in a procedural Language is a list of instructions where each statement tells the computer to do something. The focus is on processing, the algorithm needed to perform computation.

(2) **Object Oriented programming →**

The object oriented approach views a problem in terms of objects involved rather than procedure for doing it.

I. **Object →** Object is an identifiable entity with some characteristics and behaviour. eg → student, chair, table is an object.

II. **Class →** A class is a template or blue print representing a group of objects that share common properties and relationships.

Basic Concepts of OOP. →

I. Data Abstraction / Data Hiding
II. Data Encapsulation
III. Modularity
IV. Inheritance
V. Polymorphism
VI. Messaging

1. **Data Abstraction** → It refers to the act of representing essential features without including the background details. eg. You are driving a car. You only know the essential features to drive a car. eg. gear handling, use of clutch, accelerator, brakes etc. But while driving do you get into internal details of car. What is happening inside is hidden from you.

2. **Modularity** → It is property of a system that has been decomposed into a set of cohesive and loosely coupled modules.

3. **Data encapsulation** → The wrapping up of data and operations / functions into a single unit called class is known as Encapsulation. The data is not accessible to the outside world and only those functions which are wrapped in the class can access it.

Polymorphism → poly means ... morphism means forms.

It is the ability for data to be processed in more than one form. eg

If you give 5+7, it results 12, the sum of 5 and 7. but if you give 'A+B', its results into 'ABC' the concatenated strings.

Therefore, in this '+' acts differently in different enviornment.

Advantages of OOP →

1) It models real world well.
2) With OOP, programs are easy to understand.
3) OOP offers classes Reusability.
4) OOP facilitates Quick development of classes is possible.
5) With OOP, programs are easier to test, manage and maintain.

Disadvantages of OOP →

1) With OOP, classes tend to overly generalised.
2) The OOP programs design is tricky.
3) One needs to do proper planning and proper design for OOP programming.

Data hiding → Data hiding deals with visibility of the members. The level of visibility defines which parts of the code can access the member. Three levels of visibility – private, public & protected.

Overloading → 1) The process of making an operator to exhibit different behaviours in different instances is called operator overloading.

2) Function overloading → Single fxn name used to handle different numbers and different type of arguments is called fxn. overloading.

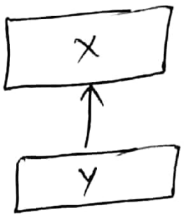
Messaging → An object oriented ~~pgm~~ program consists of set of objects that communicate with each other in order to solve a particular problem. Objects communicate with each other by sending & receiving information much the same way as people pass messages to one another. A message for an object is a request for execution of a function & therefore will invoke a function in the receiving object that generates the desired results.

eritance → Inheritance is the capability of one class of things to inherit the capabilities or properties from another class. Inheritance provide the idea of reusability. It means we can add additional features to existing class without modify it. In this, there is one base class and another is derived class.

Types of Inheritance →

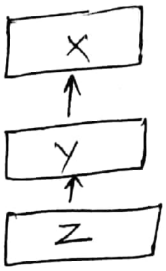
1) Single Inheritance
2) Multilevel Inheritance
3) Multiple "
4) Hierarchical "
5) Hybrid "

1) Single Inheritance →



Derivation of a single class from only one base class. The class X is the base class and Y is the derived class. further no class is derived from Y.

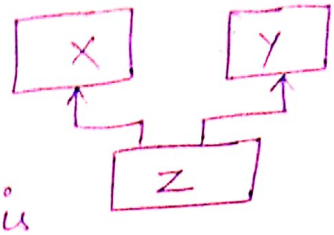
2) Multilevel Inheritance →



Derivation of a class from another derived class is called multilevel Inheritance. The class X is the top base class and Y is the derived class of X. further class Z is derived from class Y. Here, class Y is not just derived class,

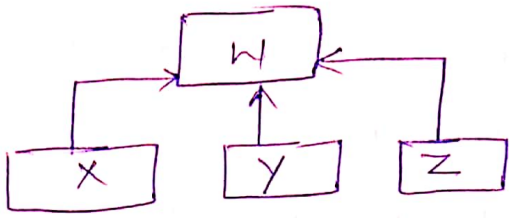
but also base class for class Z. further, can be used as base class for further der...

3, Multiple Inheritance →



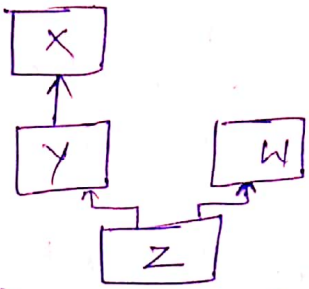
Derivation of a class from two or more base classes is called multiple Inheritance. The class X and Y are base classes and class Z is derived class. Z inherits the properties of both class X and Y. further, no class is derived from class Z.

4, Hierarchical Inheritance →



Derivation of two or more classes from a single base class. Class W is the only base class and classes X, Y, Z are derived classes. further, more of the class X, Y, Z is used as base class.

5, Hybrid Inheritance →



Derivation of a class involving two or more forms of Inheritance. In this case, multilevel and multiple Inheritances are involved. class Y is derived from class X. further, derived class Y acts as a base class. The class Z is derived from classes Y and W.

Operations on objects → The kind of operations that are performed on object comprises:

— Modifier → An operation that alters the state of an object.

1, Selector → An operation that access the state of the object but does not alter it.

3, Iterator → An operation that permits all parts of an object to be accessed in some well defined order.

4, Constructor → An operation that creates an object and or initializes its state.

5, Destructor → An operation that frees the state of an object & destroys the object itself.

Relationships among objects →

Objects contribute to the behaviour of a system by collaborating with one another. The relationship b/w any two objects encompasses the assumptions that each make about the other, including what operations can be performed & what behaviours results.

Objects communicate with each other through links, where a link defined as a physical connection b/w objects. As a participant in a link, an object may play following roles:

1, Client → that operates on other objects, but is never operated upon by other objects.

2, Server → that never operates upon other objects, it is only operated upon by other objects.

3, Agent → that can operate on other objects or operated upon by other objects. It is usually invoked to do work on behalf of an actor.

Interface of a class → The interface of a class provides outside view of a class and encompasses the abstraction while hiding its structure and the secrets of its behaviour. The interface consists of the declarations of all the operations (constants, variables) that are needed to complete the abstraction.

Implementation of class → It is its inside view that encompasses the secrets of its behaviour. It consists of implementation of all the operations defined in the interface of the class.

Unit -3

Chapter 11 Classes & Objects

Class: A class is a way to bind the data & its associated functions together. It allows data (and functions) to be hidden.

A class specification has two parts :-

1. Class declaration
2. Class function definitions

The class declaration describes the type & scope of its members. The class function definitions describe how the class functions are implemented.

General form of a class declaration :

```
class class_name
{
  private:
        variable declaration;
        function declaration;
  public:
        variable declaration;
        function declaration;
};
```

Functions & variables are collectively called class members. The keywords private & public are known as visibility labels. private member can be accessed only from within class. public members can be accessed from outside the class also. By default, all the members are private. So private keyword is optional.

Variable declared inside the class are known as data members & functions are known as member functions.

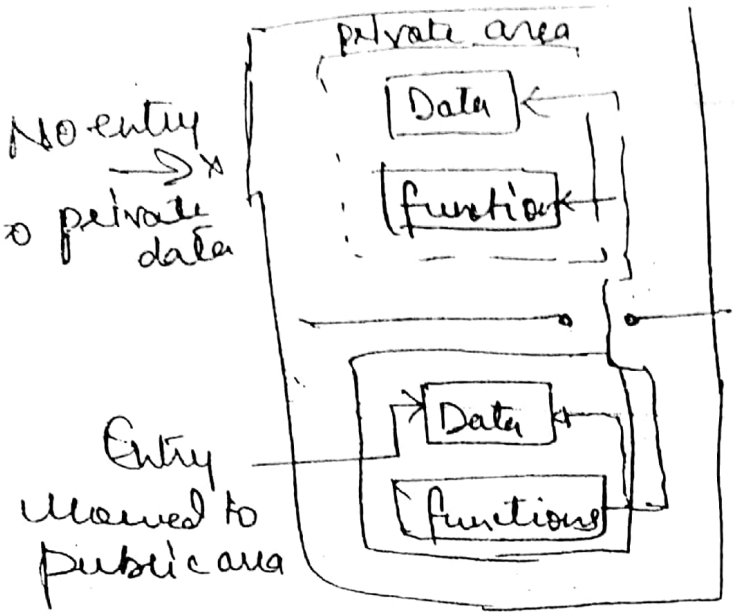


fig: Data hiding in classes

Example:

```
Class item
{
    int number;      // variable declaration
    float cost;      // private by default
public:
    void getdata (int a, float b); // function
    void putdata (void);           // declaration as
};                                 // prototype
```

Creating objects:

Once a class has been declared, variables of that type can be created using class name.

For e.g. Item x; // memory of x is created

creates a variable x of type item. In c++, class variables are known as objects. More than one object can be declared in single statement.

In another way

```
item x, y, z;
class item
{ ---
```

Accessing class members :

. Object-name . function-name (actual-args)

X · getdata (100, 75.5);

would assign the value 100 to number and 75.5 to cost.

X · putdata would display the value data members.

```
class XYZ
 { int x;
   int y;
 public:
   int z;
};
XYZ p;
 p.x = 0 // error, x is private
 p.z = 10 // Ok, z is public
```

Defining Member function

Member functions can be defined in two places

- outside the class definition.
- Inside the "

Outside: member functions that are declared outside a class a class have to be defined separately outside the class.

General form of a member function is :-

. return-type class-name :: function-name (arg. declaration)
{ function body
}

the membership label class-name :: tells that the function function-name belongs to class-name. The symbol :: is called scope resolution operator.

```
void item :: getdata (int a, float b)
  { number = a;
    cost = b;
  }
void item :: putdata (void)
{ cout << "Number: " << number << "\n";
  cout << "Cost: " << cost << "\n";
```

Characteristics of member function :-

* Several different classes can use same fun. The membership label will resolve their scope.
* Member fun. can access the private data of the class. A non-member fun. can't do so.
* A member fun. can call another member fun. directly, without using the dot operator.

Inside the class definition :

This method replaces the function declaration with the actual function definition inside the class

```
class item
  { int number;
    float cost;
  public:
    void getdata (int a, float b); // declare
    void putdata (void)
    { cout << number << "\n"
      cout << cost; ... }; };
```

Program with class :

```
#include <iostream.h>
class item
{
   int number;        // private by default
   float cost;
public:
   void getdata(int a, float b); // prototype
                                 // to be define
   // Function defined inside class
   void putdata(void)
   {
     cout << "number: " << number << "\n";
     cout << "cost: " << cost << "\n";
   }
};
// --- Member fun. definition ---
void item :: getdata(int a, float b)
{
   number = a;  // private variables directly
   cost = b;
}
// --- main program ---
int main()
{
   item x;
   cout << "\n object x " << "\n";
   x.getdata(100, 299.95);
   x.putdata();
   item y;
   cout << "\n object y" << "\n";
   y.getdata(200, 175.50);
   return 0;
}
```

Data Members → A data member of a class can be declared as static. A static member variable has certain special characteristics like -

(1) It is initialized to zero when the first object of its class is created. No other initialization is permitted.

(2) A static variable is initialized before main() fxn.

(3) Only one copy of that variable is created for entire class and is shared by all the objects of that class, no matter how many objects are created.

(4) It is visible only within the class, but its lifetime is entire program.

Static variables are normally used to maintain values common to the entire class.

eg.

```
# include <iostream>
using namespace std;
Class Test
{  static int count;
   int number;
Public:
   void getdata (int a)
   {  number = a;
      count ++;
   }
   void getcount (void)
```

```
{   cout << "count";
    cout << count << "\n";
} };
int item::count=0;
int main()
{  item a,b;
   a.getcount();
   b.getcount();
   a.getdata(100);
   b.getdata(200);
  cout << " After reading data" << "\n";
   a.getcount();
   b.getcount();
   return 0;
}
```

O/P

```
Count: 0
count: 0
After reading da
Count: 2
Count: 2
```

count is incremented whenever the data is read into an object.

Static Member fxns →

Like static member variable, we can have static member fxns. A member fxn. that is declared static has following properties:-

- A static fxn. can have access to only other static members (fxns. or variables) declared in same class.
- A static member fxn. can be called using class name as :

```
class-name :: fxn. name;
```

tatic fxn. showcount() displays the no. of objects
aled till that moment.

showcode() displays the code no. of each object.

eg →

```
# include <iostream>
# include <iomanip>
class test
{ private:
   static int count;
    int code;
  public:
    Test () {                    // constructor
      code = ++count;
    }
   static void showCount() {      // static member
  cout << "\nValue of count :" << count << endl;
   }
    void showCode() {
      cout << "\n object code :" << code << endl;
    } };
int Test :: count = 0;
  void main()
{ Test t1, t2 ;                  // create 2 objects
  Test :: showCount();
  Test t3;
  Test :: showCount();
   t1 . showCode();
   t2 . showCode();
}  t3 . showCode();
```

O/P → Value of count :

Value of count : 3

Object code : 1

Object code : 2

Object code : 3

Use of const keyword →

Constant is something that doesn't change. In C and C++ we use keyword const to make a program elements constant. Const keyword can be used in many contexts in C++. It can be used with :

1. variables
2. Class Data Members
3. Class Member fxns.

(1) **Constant variables** → If u make any variable as constant, using const keyword, you can not change its value. Also, constant variables must be initialized while declared.

```
int main( )
{ const int i = 10;
  const int j = 20;
   i++;   // compile time error.
}
```

(2) **Const class Data Members** → These are data variables in class which are made constant. They are not initialized during declaration. Their initialization

in the constructor.

```
class Test
{ const int i;
  public:
  Test ( int x)
  { i=x;
  } };
```

(3) Const class member fxn →

A const member function never modifies data members in an object.

Syntax → return type fxn. name( ) const;

eg. int f( ) const ;

```
class myclass
{ int x;
  public:
  void fun( )
  { x=10;
  }
  void cfun( ) const
  { x=200;  ⟶ error (this is constant)
  }
```

p b = 10
c = 20
d = 30

This pointer → All non-static member fxns of an object have access to a special pointer named this. It holds the address of object whose member fxn is invoked.

In other case, when argument name of a member fxns is same as that of data members, then the arguments hides the data members. So, 'this' keyword is used to resolve any namespace conflict b/w member fxn. arguments and data members. eg

```
# include <iostream>
# include <iomanip>
class sample
{ private:
    int a;
    int b;
  public:
    void setData (int a, int b)
{
    this->a = a;
    this->b = b;
};
void showData(){
  cout << "a = " << a;
  cout << "b = " << b;
    } };
void main()
{   sample s1;
    s1. showData();
    s1. setData (12,15);
    s1. showData();
}
```

Pointer to an Object → Pointers are also be used to hold address of object. Once pointer is initialized with the address of an then its members can be accessed using arrow '→' operator. eg

```
# include <iostream>
# include <iomanip>
  class test {
    int b;
    int c;
    int d;
   public:
     void setData (int x, int y, int z)
  {   b = x;
      c = y;
      d = z;
  }
     void showData()
    {  cout << "\n b = " <<b;
       cout << "\n c = " <<c;
       cout << "\n d = " <<d;
    }
  };
     void main( )
{ Test obj;          // create object
  Test * ptrToObj    // declare ptr to type Test
  ptrToObj = & obj;  // initialize pointer with
                        address of object
ptrToObj → setData (10, 20, 30);
ptrToObj → showData( );
}
```

This pointer → eg.

```
Class Sample
{   int a;
    int b;
    int c;
  public :
    void show()
    { cout << endl << " My object's address=" << this ;
    }
};
  void main()
  { Sample S1, S2, S3;
      S1. show();
      S2. show();
      S3. show();
  }
```

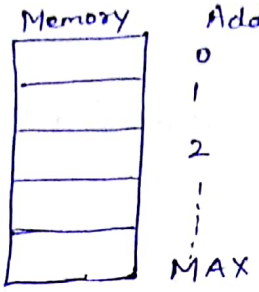
O/P

My object's address = 0x8fa6ff2
" " " = 0x8fa6ffe
" " " = 0x8fa6ff8

Chapter - 4

Pointers →

Computer's memory is organised as a linear collection of bytes or words. Each memory location is assigned a unique number called location number or address.



When you give command to run a program, the OS first finds a free block of size from main memory and loads the program into that block. Even in that block, usually program's code is loaded in one part and program in another part of the block. Each of the data operand is stored in a cell and the system associates each of the variables with these addresses.

In order to access the data operand either we use variable name or use address of the memory cell. Since these addresses are positive integer no.s, they can also be assigned to some variables and stored in memory.

Pointer → A pointer is a variable that holds the address of another variable in memory.

Variable	Value	Address
code	2125	1500
ptr	1500	2000

In this, We assign the variable code to variable ptr. So, ptr is like a pointer.

Accessing address of a variable →

To access address of a variable we use address of operator (&).

Syntax →

```
ptr = &code;
```

It will assign the no. 1500 (address of variable code) to the variable ptr.

Some restrictions must be observed :-

(b) The address of operator can only be used with variables. The following are illegal use of &.

(a) &1200 (pointing at constant expression)
(b) &(a/b) (" at expression)

(d) Pointer variable can be initialized only with use of address of operator.

```
ptr = 1500;    // illegal
```

Declaring and initializing pointers →

Like other variables, the pointer variables are to be declared first in the declaration block to tell the compiler that to which kind of values these variables will be pointing. Syntax for declaring variable is

```
type * ptr_name;
```
(* → Indirection operator)

where type is predefined data types and indicates the pointer will point to the variables of that specific data type. '*' indicates

that variable is a pointer variable.

eg int *ptr1;

Once a pointer has been declared, it must be initialized prior to its use.

Accessing a variable through its pointer → Once pointer has been assigned address of variable, then to access value of variable * operator is used.

* operator indirection

```
int *ptrX, x = 10;
ptrX = &x;
y = *ptrX;
```

∴ y = *(&x);
y = x;

Here *ptrX is a pointer variable that holds the value address of x and stores in variable y.

eg.

```
# include <iostream>
void main()
{ int *ptrX, x=10, y;
  ptrX = &x;
  y = *ptrX;
  Cout << " value of x =" << x;
  Cout << " Address of x =" << ptrX;
  Cout << " value of variable y=" << y;
}
```

O/P = Value of x = 10
Address of x = 1098
Value of y = 10

Pointer Arithmetic →

The operations that are permitted on pointers are

1. Addition of a no. to a pointer variable →

Suppose p is a pointer variable

pointing to an element of integer type like 1000, then the statement

p++ or ++p;

increment the value of p by a factor of 2 so that it points to next location that holds another value like 1002.

The increment factor will be 1 for char, 4 For long and float integer and 8 for long float

The statement -

p += i;

where i is positive integer constant or variable having +ve value, increments p such that it points to ith location beyond the location to which it is currently pointing.

2) Subtraction of a no. from a pointer variable →

Suppose p is a pointer variable, then the statement

p-- or --p;

decrements the value of p by a factor of 2, so that it now points to the location preceding current location. The statement :-

p-= i;

where i is either a positive integer constant or variable having positive value decrements p such that it points to the ith location before the location to which it is currently pointing.

Subtraction of one pointer variable from another →

One pointer variable can be subtracted from another provided both point to the same data type. The difference of two indicates the no. of bytes separating the corresponding elements.

Memory Allocation → It is imp. to note that anything that needs to be executed must be loaded into memory before it can be processed. The memory allocation is required for each program. It is done in 2 ways - statically & dynamically.

I. Static Memory Allocation →

When u give a command to run your program, the OS allocates a block of memory for your program, loads it from secondary storage (Floppy disk, hard disk, CD etc) into that block and then start its execution. When the program finishes its execution, the memory occupied by the program is automatically released. In the situation, where the amount of memory required is known before hand and the compiler then determine the memory requirements for

the data. In such, a situation, memory allocation is called static memory allocation.

2) Dynamic Memory Allocation →

The memory requirements for the instructions are always fixed. However, there may be a situation when the exact memory requirements for the data may not be known in advance or data requirements may vary from one program execution to another i.e memory requirements for the data are dynamic.

Therefore, when amount of memory is not known before hand, then it is allocated at execution time i.e when program is running. In such situation, memory allocation is called dynamic memory allocation.

3) Heap / free store → Every program is provided with a pool of unallocated memory that it can utilize during execution. This pool of unallocated memory is called as heap / free store. Whenever, the memory of large amount is required by pgm, it is taken from free store and when the previously allocated memory is not required further, it is returned back to the free store.

Dynamic Memory Management Operators / functions →

C++ provides a set of operators called dynamic memory management operators to allocate and de-allocate memory at execution time i.e. dynamically. The 2 operators are used -

1) New operator
2) Delete operator

1) New operator → It allocates the memory and always returns a pointer to an appropriate type. The new operator is defined as -

```
type * new type [size in integer];
```

eg
```
int * intPtr;
intPtr = new int [100];
```

Allocate a memory block of 200 bytes, 2 bytes for 1 integer value.

* The new operator also permits the initialization of memory locations during allocation.

The syntax for this -

```
type *ptrvar = new type ( initial value);
```

eg
```
int *intPtr = new int (100);
```

also take from user → int *p = new int; cout << *p;

How to access →

(1)
```
int *p = new int [5];
for (i = 0; i < 5; i++)
{
    p[i] = 0;
```

12, Delete operator ⇒ It is a counterpart of new operator & it deallocates (releases) memory allocated by new operator back to the free poll of memory.

Syntax ⇒ delete ptrvar;

where ptrvar can be a simple pointer variable.

However, if ptrvar is an array of pointers to objects, then we have to use delete operator as -

delete ptrvar[]; or delete[] ptrvar;

POSSIBLE PROBLEMS WITH THE USE OF POINTERS ⇒

I. Problem of Dangling / Wild pointers ⇒ These ptrs arise when an object is deleted without modifying the value of the ptr, so that still points to memory location of deallocated memory. The most common problem with pointers is that the programmer fails to initialize a pointer with a valid address. Such an initialized pointer referred to as dangling / wild pointers, can end up pointing anywhere in memory. It will take a garbage value, which can be address of any storage location. Therefore, if not properly initialized, a pointer may point to any location in memory including those locations where OS is running, your system may hang up i.e stop responding. Therefore, care must be taken to initialize pointers with valid address.

Problem of Null pointer Assignment →

One particular situation that happens is when the pointer points to address 0, which is called NULL. for eg:- This may happen that if the pointer variable is declared as global since global variables are initialized to 0. When it happens ~~and~~, then the system will display a message "Null pointer assignment" or termination of the program.

(3) Problem of memory leak →

Memory leak is a situation where the programmer fails to release the memory allocated at run time in module. When memory is allocated, a pointer variable is used to hold the address of the allocated block, however when module completes its execution, the pointer variable goes out of scope and there will be no way to reach that memory block. Therefore, care must be taken to release the allocated memory block in a module where memory was allocated.

eg. ~~else~~ int *p;

p = new int[10000];

- - -

p = new int [5000];

10000 bytes are dynamically allocated and the address of those bytes is stored in p. Later, 5000

bytes are dynamically allocated and the address is stored in p. However, original 10000 bytes have not been returned to system using delete operator.

4) Problem of Allocation failures →

Allocation failure is a situation when a program through new operator request for a block of memory, the OS could not fullfill the request of program because sufficient memory not be available in free pool of memory. In this case, new operator returns a NULL pointer indicating the allocation failure.

Chapter-5

Constructors → A constructor is a special member fxn. whose task is to initialize an object of a class when it is created. The name of constructor is same as that of class name. A constructor is automatically invoked when an object of its associated class is created. It is called constructor because it constructs object with its initial state by assigning initial values to its data members.

Characteristics of constructors →

1. They are generally declared in the public section.
2. They are invoked automatically when the objects of a given class are created.
3. They do not have any return type.
4. They can not be inherited.
5. Like other member fxns, they can be overloaded.
6. They can have default arguments.
7. Constructors can not be declared as virtual.

Need of Constructors → An array can be initialized at the time of their declaration. like

```
int abc [10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

But such initializations do not work for a class since the access for data members is usually private + protected.

This code fragment will not acceptable.

```
class student
{  private :
      int rollno;
      char name[25];
      int age;
      public :
         // member fxns
};
   Student s = { 1000, "Surbhi", 18 };
```

One possible solution for this is to write a member fxn, say init() to initialize data members —

```
class student
{  private:
      int rollno;
      char name[25];
      int age;
   public:
      void init()
      { rollno = 1000;
        strcpy (name, "surbhi");
        age = 18;
      }   // other member fxns
};
   student s;
   s.init();   // call to member fxn. to initialize data members. after creating object.
```

But if programmer forgots to do so. The object in that case might be full of garbage and might cause the program to behave in unpredictable manner.

The only solution for this problem is the use of constructor.

Declaration & Defination of Constructors

A constructor is a member fxn. whose name is same as that of class. It is placed in public section of class. It can be defined in the class itself or outside the class. Syntax →

1) Inside the class →

```
class student {
  private:
    int rollno;
    char name[25];
    int age;
  public:
    student()          // defination of constructor
    { rollno = 1000;
      strcpy(name, "subhi");
      age = 18;
    }
      // other member fxns.
};  student std;   create object, constructor is invoked implicitly.
```

Outside the class →

```
class student
{  private:
     int rollno;
     char name[25];
     int age;
   public:
     student();          // declaration
     // other member fxns
};
student :: student()
{  rollno = 1000;
   strcpy(name, "subhi");
   int age = 18;
}
student std; // create object, constructor is
                invoked implicitly.
```

Types of constructor →

1) Default constructor → In case no constructor is defined by the programmer for a class, then compiler automatically inserts one constructor in the class defination prior to compiling the class. This has following structure →

```
className() { }
```

This constructor takes no arguments and does nothing as it carries no statements. The garbage values will be assigned to the object

Zero Argument constructor is kind of constructor that takes no arguments. It is used to initialize the every created object to a same specific state i.e same initial values for all objects. eg

```
class Complex
{ private:
    float realpart;
    float Imaginarypart;
public:
  Complex()          // zero argument constructor
 { realpart = 0.0;
   Imaginarypart = 0.0;
 }
   // other member fxns
};  void main()
    { Complex n1, n2, n3;
      }
```

The data members of all 3 objects are initialized to zero value.

3) Parameterized Constructor → The constructor which take arguments is called parameterized constructor. This is used to initialize the data

members with different values when they
created. eg → parameterized constructor with explic
and implicit call

```
# include <iostream>
# include <iomanip>
  class ComplexNumber
{ float realPart;
  float ImaginaryPart;
 public:
  ComplexNumber (float real, float imaginary)
{  realPart = real;
   ImaginaryPart = imaginary;
}
   void display( )
    { cout << realPart << " +j " << ImaginaryPart << endl;
    }
};
   void main( )
{ ComplexNumber C = ComplexNumber (1.25, 2.75);
  // object created with explicit call to constructor
  Comp C.display( );
  ComplexNumber C1 (7.5, 12.25);   // Implicit call to constructor
  C1.display( );
}
```

O/P 1.25 + J2.75
7.50 + j 12.25

creates object

Copy Constructor → A constructor that accepts a reference to its own class is called a copy constructor. A copy constructor is used to initialize an object from another objects of same class. eg

```
class Complex
{  private :
       float realPart;
       float ImaginaryPart;
   public :
       Complex( Complex &cn)        // parameterized constructor that takes reference to an object of its own class as its argument
{
   realPart = cn.realPart;
   ImaginaryPart = cx.ImaginaryPart;
}
       // other member fxns
};
```

* A copy constructor is used to declare and initialize an object from another object of its class.

eg
```
Complex second(first);
        OR
Complex second = first;
```

Overloaded Constructors →

The constructors with the same name but different no. of arguments is called overloaded constructors. eg

```
class Complex
{   private:
        float realPart;
        float imaginaryPart;
    public:
      Complex( )          // zero argument constructor
      {  realPart = 0.0;
         imaginaryPart = 0.0;
      }
// parameterized constructor
      Complex( float real, float imaginary)
      {  realPart = real;
         imaginaryPart = imaginary;
      }
      Complex( Complex &cr)
      {  realPart = cr.realPart;
         imaginaryPart = cr.imaginaryPart;
      }
         // other member fxns
};
```

Program to implement the overloading of multiple constructors

```
# include <iostream>
# include <iomanip>
 class ComplexNumber
{ private:
    float realPart;
    float ImaginaryPart;
public:
    ComplexNumber()          // zero argument constructor
  {  realPart = 0.0;
     ImaginaryPart = 0.0;
  }
    ComplexNumber(float real, float imaginary)  //parameterised
{     realPart = real;
     ImaginaryPart = imaginary;
  }
    ComplexNumber (ComplexNumber &cn);  // Copy constr.
  {  realPart = cn.realPart;
     ImaginaryPart = cn.ImaginaryPart;
  }
    void display()
  { cout << realPart << " +j " << ImaginaryPart;
  }
};
    void main()
{   ComplexNumber c;
    c.display();
    ComplexNumber c1 (70.5, 120.25);
      c1. display();
```

```
ComplexNumber c2 (c1);   // reference to o
                         its argument.(
c2.display();
}
```

O/P
```
0.00 +j0.00
70.50 +j 120.25
70.50+j120.25
```

Explicit Constructors → Consider class definition

```
class Sample
{ private:
    int a;
  public:
    Sample (int x)
    { a = x;
    }
    // ---
};
```

The constructor for Sample takes one parameter. In order to create an object of Sample class we write:

```
Sample obj(5);   OR   Sample Obj = Sample(5);
```

The same object also be created by using statement

```
Sample obj = 5;
```

This statement will automatically be converted into a call to the Sample constructor with 5 being argument.

This type of implicit conversion happens only for a constructor with single argument.

If you do not want this implicit conversion to happen, you can prevent it by prefixing the keyword explicit before the single argument constructor as -

```
class Sample
{ private :
    int a;
    public :
        explicit { Sample (int x)
        { a = x;
        }
        // ----
};
```

Then if we write Sample obj = 5;
This will not be allowed. It will cause a syntax error.

Dynamic Constructors → Constructors can also be used to allocate memory while creating objects. This will allow the system to allocate the desired amount of memory for each object when the objects are not of same size, thus resulting in saving of the memory. Allocation of memory to objects at the time of their construction is called dynamic constructor of objects. The memory is allocated with new operator.

Destructors → A destructor is a special member function that is used to release the resources held by the object before the object is destroyed. It also has same name as that of class but is preceded by Tilde character (~)

eg
```
~ Complex()
{
==
}
```

Characteristics of Destructors →

(1) They are declared in the public section.
(2) They are invoked automatically when the objects go out of scope.
(3) They do not have any return type.
(4) They can-not be inherited.
(5) They can't be overloaded.
(6) Destructors can be declared as virtual.
(7) It takes no arguments.

Need of Destructors → During construction of an object by the constructor, resources may be allocated for use. Constructor may have opened files, may have created a connection to another computer on the network or may have allocated memory dynamically. These allocated resources must be deallocated before the object is destroyed. This can be done by using destructor. It is automatically invoked.

Declaration and Defination of Destructors →

Destructor can be defined inside and outside the class like -

Destructor defined inside the class →

```
class Sample
{  private:
     // data members
   public :
     ~ sample ()
     {
        // statements for destructor
     } // other member fxns
};
```

Outside the class →

```
class Sample
{  private:
     // data members
   public:
   ~Sample ();    // declaration
     // other member fxns
};
sample :: ~Sample ()
  {
     ——  // statements
  }
```

Constructors & Destructors with static members

The static data member is useful when all the objects of the same class must share a common piece of information. Memory is allocated for static variables when the class is loaded & only one copy of class variables is kept irrespective of no. of objects of that class. Every object has access to class variables. It is initialized before main().

Use → It is used to keep track of no. of objects of a class.

The constructors & Destructors being member fxns, can access static data members like –

```
#  include <iostream>
#  include <iomanip>
class  Sample
{  static int count;
   int x;
   int y;
  public:
    Sample ( ) {  // constructor
    count ++;
    cout << " \n Including newly created object "
         << " object count = " << count << endl;
  }
~ Sample ( )      # Destructor
  { count --;
cout << " Excluding object being destroyed " << " object
        count = " << count << endl;
```

```
};
int sample::count = 0;
void main()
{
    sample s1, s2, s3;
}
```

o/p

Including newly created object, object count = 1
" " " " " " 2
" " " " " " 3

Excluding object being destroyed, object count = 2
" " " " " = 1
" " " " " = 0

Initializer list → The most common task of a constructor is to initialize data members. This initialization takes place in the body of the constructor function. There is another approach of doing so; where initialization takes place following the function declarator but before the function body. A colon is used to precede the initializer list. The value is placed in parentheses following data member name.

Syntax →

```
class A
{ private:
    int a;
    int b;
    int c;
```

```
public :
   Example( int i, int j, int k): a(i), b(j), c(k)
    { }
    void show( )
    { // ...
      }
};
```

When we create object like

```
A obj(1, 2, 3);
```

1, 2, 3 passed as arguments. This happens even before constructor start executing.

This approach is useful in const data members. A const data member can not initialized in body of the constructor. So this method is used.

Program →

```
class Example
{ const float pie;
    int k;
 public:
  Example (float a, int b): pie (a)
 { k = b;
   }
 void show()
 { cout << "\n Pie =" << pie << "\n k =" << k << endl;
};
};
void main( )
```

```
{ Example obj (3.142, 12);
  obj.show();
}
```

Unit - 6

Operator Overloading & Type Conversion

Operator Overloading → It is an imp. feature in C++.

It is a type of polymorphism in which an operator is overloaded to give user defined meaning to it. Overloaded operator is used to perform operations on user defined data type. eg '+' operator can be overloaded to perform addition on various data types like for integer, String etc.

Syntax →

```
Return type operator op (arglist)
{
    // fx. body defining task
}
```

(keyword: operator; op: operator; arglist: list of arguments)

Operator function is either non-static member fxn. or friend function. The diff. b/w them is — friend function will have only one argument for unary operators and two for binary operators while a member fxn. has no argument for unary operators and only one for binary operators.

The process of overloading involves following steps —

(2)

1, Create a class that defines data type to be used for overloading an operator.

2, Declare the operator fxn: op (arglist) in public section of class. It can be a member fxn or friend function.

3, Define the operator function to implement the required operation.

Rules for overloading operators →

1, Only existing operators can be overloaded, New operators can not be created.

2, The overloaded operator must have at least one operand of user defined data type.

3, We should not change the meaning of operator for their behaviour for built-in data types. We can't make '+' to subtract one value from the other.

4, The overloaded operators follow the syntax rules of the original operators. They can not be overridden.

5, The following operators can not be overloaded.

Operator	Description
size of	Size of operator
.	Membership op
.*	Pointer to member operator
::	Scope resolution
?:	Conditional operator

friend function can not be used to overload the following operators but member fxns can be used. (3)

=	Assignment operator
( )	function call "
[ ]	subscripting "
→	Class member access "

Overloading of unary operators →

1) Overloading of Increment & Decrement operators →

When increment (++) & decrement (--) operator is used as prefix, its value is first incremented and decremented and then used.

If it is used as postfix, first its value is used and then incremented & decremented.

Syntax

```
operator ++()      // prefix notation
operator ++ (int)  // postfix "
```

↓ to indicate the compiler that operator is being overloaded for postfix.

Program → Overloading of ++, --

```
# include <iostream>
# include <iomanip>
  class counter
{ int counterValue;
  public:
```

(4)

```
Counter ( )
{   counterValue = 5;
    cout << "\n Counter's initial value =" << counterValue
}
void operator ++ ( )          // overloading of ++ for prefix
{   ++ counterValue;
}
void operator ++ (int)        // postfix notation
{   counterValue ++ ;
}
void operator -- ( )          // overloading of -- for prefix
{   --counterValue ;
}
void operator -- (int)        // postfix notation
{   counterValue -- ;
}
void showCounter( )
{   cout << "\n Current counter value = "
        << counterValue << endl;
}
};  void main ( )
    {   Counter count;
        ++ count ;                  // increment counter
        Count . showCounter ( );
        Count ++;                   //  "      "
        Count . showCounter ( );
        -- count ;                  // decrement count
        Count . showCounter( );
        Count -- ;
```

```
Counter.showCounter();
}
```

O/P
```
Counter's Initial value = 5
Current counter value = 6
 "      "        "    7
 "      "        "    6
 "      "        "    5
 "      "        "
```

2) Overloading of unary minus operator

```
class ComplexNumber
{ float real;
  float imaginaryPart;
  public:
  ComplexNumber(float real, float imaginary)
  { realPart = real;
    imaginaryPart = imaginary;
  }
    void operator -()          // overloading of -
    { realPart = -realPart;
      imaginaryPart = -imaginaryPart;
    }
    void display()
  { cout << realPart << "+" << imaginaryPart << "j";
  }
};    void main()
      {
        ComplexNumber number(1.5, 2.5);
        cout << "\n Original no.";
```

(6)

```
number. display();
-number;
cout << "\n complex no. after negate operation";
number. display();
}
```

O/P

```
Original complex no. : 1.50 +2.50j
Complex no. after negate : -1.50 - 2.50j
```

Overloading of binary operators →

```
# include <iostream>
# include <iomanip>
class complex
{ int a, b;
  public:
   void getvalue()
    { cout << "Enter the value of complex nos a,b";
      cin >> a >> b;
    }
   complex operator + (complex ob)
    { complex t;
      t.a = a + ob.a;
      t.b = b + ob.b;
      return (t);
    }
   complex operator - (complex ob)
    { complex t;
      t.a = a - ob.a;
      t.b = b - ob.b;
      return(t);
    }
```

```
void display()
{ cout << a << "+" << b << "i" << "\n";
}
};
void main()
{ complex obj1, obj2, result, result1;
  obj1.getvalue();
  obj2.getvalue();
  result = obj1 + obj2;
  result1 = obj1 - obj2;
  cout << "Input values"
  obj1.display();
  obj2.display();
  cout << "result";
  result.display();
  result1.display();
}
```

o: display

a=4
b=4

O/P.

```
Enter the value of Complex nos a,b
4 5
Enter the value of Complex nos a,b
2 2
Input values
4+5i
2+2i
Result
6+7i
2+3i
```

Type Conversion → It is a process of converting basic data type to user-defined data-types or vice versa. 3 types.

1) Conversion from basic type to class type.
2) Conversion from class type to basic type.
3) Conversion from one class type to another class type.

1. Basic type to class type ⇒ When a value of basic type is assigned to an object, the constructor is implicitly invoked and automatic conversion takes place.

eg.
```
class Time
{ private:
    int hours;
    int minutes;
    int seconds;
  public:
    Time( int timeinseconds)
    { hours = timeinseconds/ 3600;
      timeinseconds %= 3600;
      minutes = Timeinseconds/60;
      timeinseconds %= 60;
      seconds = timeinseconds;
    }
    void display( )
{ cout << hours;
  cout << minutes;
  cout << seconds;
}
void main( )
{ Time t1 = Time(3738);
  Time t2 = 3738;
}
```

Conversion from class type to basic type →

Constructors do not support this conversion.

C++ allow us to define an overload cast operator fxn. usually referred to as conversion fxn. that can be used to convert class type to basic type.

Syntax → operator typename( )

```
{  // statements
}
```

Cast operator fx. has characteristics →

1) It must be a class member.
2) It has no return type.
3) It has no arguments.

eg →

```
class data
{ int x;
  float f;
  public:
   data (int n, float m)
   { x=n;
     f=m;
   }
operator int ()                    // cast operator
   { return x;
   }
operator float()
   { return f;
   }
```

(10)

```
void show()
    { cout << " value of x, f = " << x << ", " << f
};
int main()
  { int ac;
    float bc;
    data a (100, 25.25);
    a . show();
     ac = a;        // convert class type to int
     bc = a;        //  "    "    "  "  to float
     cout << " int value = " << ac << endl;
     cout << " float value = " << bc;
      return 0;
    }
```

O/P → value of x, f = 100, 25.25
int value = 100
float value = 25.25

3. One class to another class type →

In this we convert one class type to another

like obj X = obj Y;

obj X is object of class X. obj Y is object of class Y.

The class Y type data is converted to the class X type data and the converted value is assigned to obj X. Since, conversion takes place from class Y to class X. Then

Y = Source class X = Destination class

Such conversions b/w objects of different classes

can be carried out by either a constructor or by a conversion fxn.

We know that conversion fxn.

operator typename ()

converts class object of which it is member to typename. Here typename refers to destination class. Therefore when a class needs to be converted, a casting fxn. can be placed in source class and result is given to the destination class object.

In one-argument constructor, the argument belongs to the source class is passed to the destination class for conversion. The conversion constructor be placed in the destination class.

eg of casting fxn. operator

```
class circle
{ int radius;
  public:
  circle (int r)
  { radius = r;
  }
  void show ()
  { cout << radius;
  }
};

class rectangle
{ private:
    int length, breadth;
  public:
  rectangle(int l, int b)
  { length = l;
    breadth = b;
  }
  operator circle()
  { return circle(breadth);
  }
  void show()
  { cout << length;
    cout << breadth;
  }
};

int main()
{
```

(12)

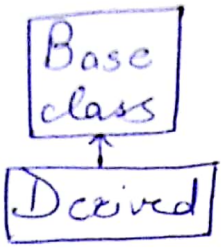
```
rectangle r (20,10);
circle c (15);
c = r;          // conversion from one class type to another
c.show();
r.show();
}
```

O/P → 10
20
10

Unit-7

Inheritance

Inheritance is process of creating new classes from existing class by inheriting some properties and add some extra features by it's own existing class is called base class and new class is called derived class.



Forms of inheritance → Explanation in notes of Unit - 1

Defining derived classes →

Syntax for derived class →

```
class DerivedClassName : [Access Specifier] BasicClass Name
{ // members of derived class
};
```

Access Specifier is optional & can be public, private or protected. By default is private.

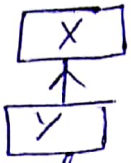
Role of Access Specifier :-

(1) When public access specifier is used, then public members of base class remain public members of derived class & same for protected.

(2) When private access specifier is used, then public & protected members of base class become private members of the derived class.

(3) When protected access specifier is used, the public members of base class become protected members of derived class & protected members of base class remain protected members of derived class.

(1) Single Inheritance →



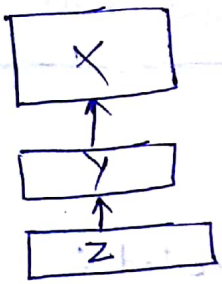
When only one class is derived from single base class.
Derived class is not used as base class for another class derivation.

```
#include<iostream>
#include<iomanip>
#include<string>
class X          //base
{protected:
  char name[50];
  int age;
  public:
  void display()
{cout<<"base class";
}
};
class Y : public X     //Derived Class
{ private:
    int weight;
  public:
  void getdata()
{cout<<"Enter name:";
cin>>name;
cout<<"Enter age:";
cin>>age;
```

```
cout << "Enter weight : ";
cin >> weight;
}
void showData()
{ cout << "Name : " << name;
  cout << "\n Age : " << age;
  cout << "\n Weight : " << weight << endl;
};
void main()
{ Y objY;
  objY.getData();
  objY.showData();
  objY.display();
}
```

O/p → Enter Name : Riti
Enter age : 12
Enter weight : 30
Name : Riti Age : 12 weight : 30

2. Multilevel Inheritance →

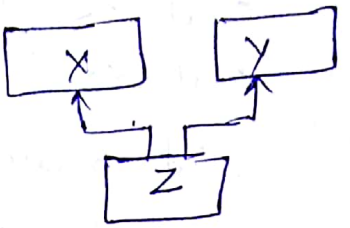


eg →

```
class X
{ protected :
    char name[30];
    int age;
};
class Y : public X
{ protected :
    int address;
};
class Z : public Y
{ private :
    int weight;
  public :
    void getData(void)
    { cout << "Enter name";
      cin >> name;
      cout << "Enter age";
      cin >> age;
      cout << "Enter address";
      cin >> address;
      cout << "Enter weight";
      cin >> weight;
    }
    void show()
    { cout << "\n name : " << name;
      cout << "age : " << age;
      cout << "\n addres " << addres;
      cout << "\n weight " << weight;
```

```
};
void main()
{ Y ob;
  ob.getData();
  ob.show();
}
```

3. Multiple Inheritance →




```
class X
{  protected :
     int x;
   public :
     void showX()
   { cout << "\n X::x =" << x << endl;
   } };
class Y
{  protected :
     int y;
   public :
   void showY()
   { cout << "\n Y::y =" << y << endl;
   } };
class Z : public X, public Y
{  private :
     int z;
   public :
   void setData(int i, int j, int k)
   {  x = i;
      y = j;
      z = k;
   }
   void showZ()
   { cout << "\n Z::z =" << z << endl;
   }
};
void main()
{ Z objz;
  objz.setData(5, 10, 15);
  objz.showX();
  objz.showY();
  objz.showZ();
}
```

O/P

```
X::x = 5
Y::y = 10
Z::z = 15
```

Ambiguity in multiple Inheritance →

When the base classes have members with same name, while the classes derived from these classes have no member with that name, then accessing that common member causes ambiguity. Then the compiler will not able to figure out which of these members should be used.

eg →

```
class X
{ public:
    void show()
{ cout <<" \n class X \n";
 } };
class Y {
  public:
    void show( )
  { cout << "\n class Y \n";
  } };
class Z : public X, public Y
 { };
  void main( )
{ Z objz ;
   objz. show( ); // ambiguity
}
```

When this pgm is compiled, compiler will give error msg like:

Member is ambiguous: 'X:: show' and 'Y:: show'

This ambiguity can be resolved using scope resolution operator to specify the class in which fun lies-

Thus,

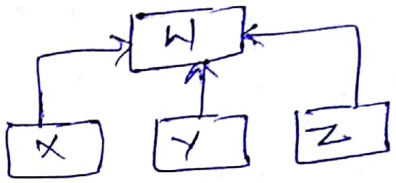
```
objz.X::show();
```

refers to version of show() of class X.

```
objz.Y::show();
```

for class Y;

4) Hierarchical Inheritance →

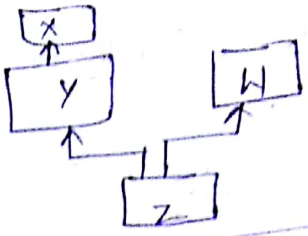



```
class W
{ public:
    void showW()
    { cout <<"\n Base class
      W \n";
    } };
class X: public W
{ public:
    void showX()
    { cout <<"\n Derived class
      X \n";
    } };
class Y: public W
{ public:
    void showY()
    { cout <<"\n Derived class
      Y \n";
    } };
class Z: public W
public:
    void showZ(){
      cout <<"\n Derived class Z \n";
} };
void main()
  { W objW;
    X objX;
    Y objY;
    Z objZ;
   objW.showW();
   objX.showX();
   objY.showY();
   objZ.showZ();
  }
```

O/P

```
Base class W
Derived class X
"           Y
"           Z
```

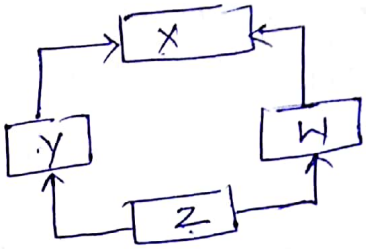
Hybrid Inheritance →




```
class X
{ public:
    void showX()
  { cout << "\n classX \n";
  }
};
class Y : public X
 { public:
     void showY()
   { cout<<"\n class Y \n";
   }
};
class W
 { public:
    void showW()
   { cout << "\n classW \n";
    }
};
class Z : public Y, public W
   { public:
      void showZ()
     { cout << "\n class Z \n";
      }
};
   void main()
   { Z objz;
     objz.showX();
     objz.showY();
     objz.showW();
     objz.showZ();
   }
```

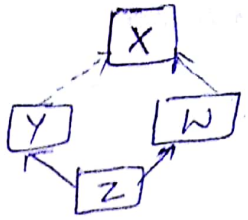
(B) Multipath Inheritance →

When a class is derived from two or more classes that in turn are derived from same base class.




```
class X {
 public:
   void showX()
   { cout<<"\n class X \n";
   }
};
class Y: public X
 { public:
    void showY()
   { cout<<"\n class Y \n";
   }
};
class W : public X
{ public:
    void showW()
   { cout <<"\n class W \n";
   }
};
```

```
class Z : public Y, public W
{ public:
    void showZ()
    { cout << "\n class Z \n";
    }
};
void main()
{ Z objz;
  objz.showY();
  objz.showW();
  objz.showZ();
}
```




```
eg→ class X
    { public:
        void show()
        { cout << "\n class X
        }
};
class Y: virtual public X
class W: virtual public X {
class Z: public Y, public W
    }
  void main()
{ Z objz;
  objz.show(); // No ambig
}
```

Virtual Base Class →

When more than one class derives from the same class, these derived classes are further used in the derivation, prefix the keyword virtual before the access specifier in the class derivation. With virtual derivation, only single shared copy of the base class object is inherited regardless of how many times the base class occurs within the derivation hierarchy. The shared base class object is called virtual base class.

Object Slicing →

When an object of, derived class is assigned to an object of base class, only the base portion of object is assigned, portion corresponds the derived class is sliced off. This phenomenon is called object slicing.

cin >>

```
class A
{ int a;
  public:
    A(int x)
    { a = x;
    }
    void show()
  { cout << "\n a =" << a << "\n";
    }
};
 class B : public A
   { private:
       int b;
     public:
       B(int r, int s) : A(r)
     {   b = s;
       } void show()
         {  cout << "\n b = " << b << "\n";
         }
   };
   void main()
 {  A  objA(10);
    B  objB(12, 25);
    objA = objB;
    objA.show();
 }
```

o/p a = 12

Overriding member functions →

The derived class can have a member functions with same name, same return type and same list of arguments as defined in the base class. The member function of the derived class overrides the member function of the base class.

eg →

```
class X
{ protected :
    int a;
    float b;
  public:
    void read( )
    { cout <<"\n Enter a = ";
      cin >> a;
      cout << "\n Enter b = ";
    }  cin >> b;
  Void display( )
  { cout <<"\n a = "<<a;
    cout <<"\n b = "<<b <<"\n";
  }
};
class Y : public X
{ protected :
    int c;
  public:
   void read( )
   { X:: read( );
     cout << "Enter c = ";
     cin >>c;
   }
   void display( )
   { X:: display( );
   } cout <<"c = "<<c<< "\n";
};
void main( )
{ Y objY;
  objY. read( );
  objY. display( );
}
```

O/P → Enter a = 5
Enter b = 2.75
Enter c = 10

a = 5
b = 2.75
c = 10

(11) **Object Composition** → There are some situations objects are used as data members in a class. The mechanism of using objects in a class as data members is referred to as object composition.

⇒ Relationship b/w objects, where one object owns or has the other object. eg Car has or owns Motor

→ When car is build, its motor is built also.

→ When car is destroyed it's motor is destroyed.

∴ In case of inheritance, the constructors of base classes are invoked before the constructor of derived class. But in composition, constructor of derived class is invoked first.

Syntax →

```
class B
{ //
};
class D
{ //
  B objB;
 public:
  D (arg-list1) : objB (arg-list2)
  { // ordinary member Initi-
    - alization
  }
};
```

example →

```
class B
{ private:
    int a;
  public:
  B ( int i)
  { cout << "\n constructor B
    is invoked \n";
    a = i;
  }
  void showB ( )
    { cout << a;
    }
};

class D
{ int d;
  B objB;
 public:
D ( int r, int s): objB(r)
{ cout << "\n Constructor D
  is invoked";
  d = s; }
```

```
void showD( )
{ cout << "\n object member of class D = ";
  objB.showB( );
  cout << "\n Ordinary member of class D = ";
  cout << d << "\n";
} };
  void main( )
{   D objD(10,20);
    objD.showD( );
}
```

O/P
Constructor B is invoked
Constructor D is invoked
object member of class D = 10
Ordinary member of class D = 20

Delegation → Delegation is like Inheritance done manually through object composition. It is way of making object composition as powerful as inheritance for reuse.

```
class Publication
{ // body
};
class Sales
{ // body
};
class Book : public Publication, public sales
{ // body of book class
}
```

The above functionality can also be achieved by composing objects of classes Publication and Sales as data members of book class -

```
class Books
{ //
    publication pub;
    Sales sal;
//
};
```

eg→
```
class Publication
{ private:
    char title[40];
    int price;
public:
    void getdata()
    { cout<<"Enter title:";
      cin.getline(title, 40);
      cout<<"Enter price";
      cin>>price;
    void showData()
    { cout<<"Title:"<<title;
      cout<<"\n Price"<<price;
    }
};
class Sales
{ int salefigures[3];
  public:
  void getData()
{ for(int i=0; i<3; i++)
{ cout<<"Enter sales of Month"
  <<i+1<<":";
  cin>>salefigures[i];
}
}  void showData()
   {
```

```
int totalSales=0;
for(int i=0; i<3; i++)
{ cout<<"\n Sales of Month
<<i+1<<"="<<salefigures[i];
totalsales += salefigures[i];
}
cout<<"\n Total Sales="<<
 totalSales;
}
class Book
{ int pages;
    Publication pub;
    Sales sal;
 public:
  void getdata()
  {pub.getdata();
   sal.getdata();
  }
  void showData()
  { pub.showData();
    sal.showData();
  }
};
  void main()
{ Book Bookobj;
   Bookobj.getData();
   Bookobj.showData();
}
```

Constructors play an important role in initializing an object's data members. The order of execution of constructors is as -

(1) Base class constructor → If there is more than 1 base class, the constructors are executed in the order the base classes appear in the class derivation list. However, if there is virtual base class in the class derivation list, its constructor will be executed first irrespective of its position in the derivation list.

(2) If there is more than one member class the constructors are executed in the order t objects are declared in the class.

3. Derived class constructor.

Order of execution of Destructors →

Unlike constructors, destructors in the class hierarchy are invoked in the reverse order of constructor invocation. The destructor of that class whose constructor was executed last, while building the object of derived class, will be executed first i whenever the object of the derived class goes out of scope.

13

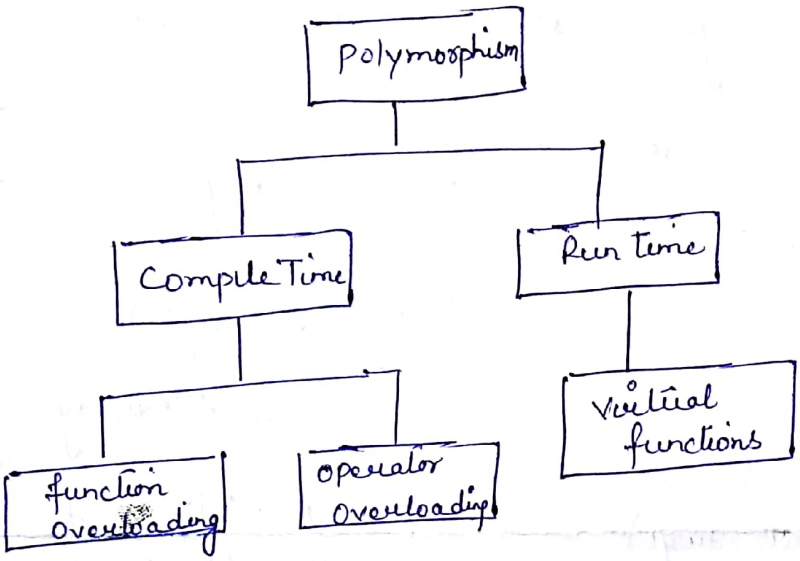
program to demonstrate order of execution of constructors and Destructors →

```
class A
{
private:
  int a;
public:
  A()
{ a = 1;
cout << "\n Constructor of
        class A \n";
}
 A()
cout << "\n Destructor of
class A \n";
};
class B
{ int b;
  public:
   B()
{ b = 2;
  cout << "\n Constructor of Class
      B \n";
}
~B()
{ cout << "\n Destructor of
   class B \n";
};
class C
{ int c;
  public:
    C()
  { c = 3;
  cout << "\n Constructor of
  class C \n";
}
~C()
{ cout << "\n Destructor of class C";
};
class D : public A, virtual Public B
{ int d;
  public:
   D() {
     d = 4;
    cout << "\n constructor of class
        D \n";
}
  ~D() {
cout << "\n Destructor of class D";
};
void main()
{ D objD;
}
```

o/p
Constructor of class B
" " A
" " C
" " D
Destructor D
" " " A C B

Chapter-8 Virtual functions and polymorphism

Polymorphism → It is another Imp. feature of OOPS. It is combination of 2 words - poly and morphism. The word 'poly' means many and 'morphism' means forms. It means many forms.



Binding → It is the process of resolving the fxn. call i.e. linking the function call to its definition.

Two types -

1) Binding at compilation time is called early binding or static binding.
2) Binding at execution or runtime is called late binding or dynamic binding.

① **Static / Early binding** ⇒ The information about the no. and type of arguments

(2) is available to the compiler at the time of
Therefore, compiler is able to select the appr.
fxn. for particular call at the compile time it...
This is called early binding. It is early in the
sense that fxn. definition is bound to its call
at the compile time and static means binding
can't be changed at execution time.

eg.

```
class BaseClass
{ protected:
    int d;
  public:
   BaseClass() { }
   BaseClass(int arg)
{   d = arg;
  }
  void display()
  { cout << endl << "Display
  method of base class";
cout << endl << "d =" << d << endl;
  } };
class Derived : Public BaseClass
{ int k;
 public:
   Derived() { }
Derived( int arg)
{   k = arg;
 }
 void display()
{ cout << endl << " Display method of
                    derived ";
cout << endl << "d =" << d << endl;
cout << endl << "k =" << k << endl;
 } };
void main()
{ BaseClass varB(10);
   varB.display();
  Derived varD(20);
   varD.display();
}
```

O/P
Display method of base Class
d = 10
Display method of derived
d = 10
k = 20

In this pgm, all calls to fxns are resolved at compile time. (3)

2) Late binding → (Dynamic binding) → The fxn. is linked with particular class after the compilation i.e during program execution, the process is called Late or dynamic binding. It is Late in the sense that a fxn definition is bound to its call at execution time and dynamic in the sense that this binding can be changed at execution time. This is also called runtime polymorphism.

Runtime polymorphism is achieved by following steps -

1) Preceding the member fxn. with keyword virtual in the base class.
2) Accessing member fxn. through a pointer of base class.

eg-

```
class Base
{ int d;
  public:
    Base ( ) { }
Base ( int arg )
{ d = arg ;
  }
  virtual void display ( )
{ cout << "Display method of base class";
cout << " d = " << d << endl;
  } };
```

```
class Derived : public Base
{ int k;
  public:
  Derived ( ) { }
Derived ( int arg )
{ k = arg;
 }
 void display ( )
 { cout << " Display method of derived class";
cout << "d = " << d << endl;
cout << "k = " << k << endl;
```

```
};
(4) void main()
{   // declare pointer to base class
    Base * baseptr;
    Base varB(10);
    baseptr = &varB;        // assign address of base class
                               to pointer.
    baseptr -> display();
    Derived varD(20);
    baseptr = &varD         // assign address of derived cla
                               to pointer
     baseptr -> display();
}
```

O/P →
```
Display method of base class
d = 10
Display method of derived class
d = 10
k = 20
```

The virtual keyword tells the compiler that binding for this member fxn, whether the fxn is called on object of base class or on object of derived class be delayed till the runtime.

Virtual functions → A member fxn. whose function declaration is preceded by virtual keyword is known as a virtual fxn. These fxns. are defined in a base class in the public section and they provide a mechanism by which the derived classes can override it. The syntax for defining a virtual fxn. is

```
class Sample
{ private:        // data members of class
  public:
        // fx. members of class
  virtual Returntype functionname(arguments)
  {   // body of virtual fxn.
  }
};
```

At runtime, it allows to decide w/c overridden form of the fxn. is to used based on type of object pointed by base pointer.

Pure Virtual fxns → Virtual fx., normally serves as a framework for future design of the class hierarchy as they can be overridden in the derived classes. In many cases, these virtual fxns. are declared without any body i.e. they

don't have any definition. Such virtual fn are called pure virtual fxns. These fxns. are also called do-nothing fxns.

Syntax –

```
class Sample
{  private :
        // data members of class
   public :
   virtual returntype functionname (arguments) =0;
};
```

Abstract classes → Abstract class is a class which contains at least one pure virtual function in it. Abstract classes are used to provide an interface for its sub classes. Classes inheriting an abstract class must provide definition to pure virtual fxn., otherwise they will also become abstract class.

Characteristics → (1) We can't create object of abstract class, but pointers, references of abstract class type can be created.

(2) Classes inheriting an abstract class must implement all pure virtual fxns or else they will become abstract too.

```
class Base
{ public:
   virtual void show() = 0;     // pure virtual fxn.
};
class Derived: public Base
{ public:
    void show()
   { cout << " Implementation of virtual fxn";
   }
};
int main()
{ Base *b;
   Derived d;
    b = &d;
    b-> show();
   }
```

O/P →
Implementation of virtual fxn.

Virtual Destructor → In C++, a destructor is generally used to deallocate memory for a class object and its class members whenever an object is destroyed.

. A problem can occur when using polymorphism to process dynamically allocated objects of a class of hierarchy.

→ If an object with non virtual destructor is destroyed explicitly by applying a delete operator to a base class pointer to the object, then only base class

⑥ destructor is only called on the object. Thus, need for virtual destructor is required and it is explained by example -

```
# include <iostream>
 class Base
{ public:
    Base ()
  { cout <<" Constructing base";
  }
~ Base()
{ cout <<" Destroying base"; }
};
class Derived : public Base
{ Derived ( )
  { cout << " Constructing derived";
  }
 ~ Derived( )
{ cout <<"Destroying Derived";
}
};  void main( )
  { Base * baseptr = new Derived( ); // upcasting.
      delete baseptr;
  }
```

[It uses super class's reference or pointer to refer to a subclass object].

O/P

```
Constructing base
Constructing derived
Destroying base
```

In above example, we can see that constructors get called in the appropriate order. But there is problem that destructor for Derive class does not get called at all when we delete baseptr. To fix this problem, we can make the base class destructor virtual, that will ensure that destructor for any class that derives from Base will be called.

eg. of virtual Destructor →

We need to change the destructor in the Base class and by using virtual keyword so that destructors are called in appropriate order like.

```
class Base
{ public:
    Base()
    { cout << "constructing base"; }
      virtual ~Base()
    { cout << "Destroying base";
    } };
```

Unit - 4

Exception Handling

Exception → Exceptions are abnormal conditions that arise in a particular code of a sequence of a program at runtime. Basically, it is runtime error. eg a no. is divided by zero, running out of memory, Array Index out of bound exception etc.

Exception Handling → Exceptions are handled so that part of program which is correct should atleast run. This process is called exception Handling. This improves program readability and modifiability.

Mechanism of Exception Handling →

It involves following steps -

1) find the problem (Hit the exception)
2) Inform that an error has occured (Throw the exception)
3) Receive the error information (Catch the exception)
4) Take corrective actions (Handle the exception)

Keywords Used for Exception Handling are -

1) Try 2) Throw 3) Catch

1) **Try and Catch block** → Try is the block of

statement which we want to keep under monitoring. The exception which is thrown out of try by using throw block is catch by immediately following the catch block.

In catch block, we mention the type of exception which we want to catch.

Syntax →

```
void main()
{ try
  { // ---- statements w/c cause, detect exception
    throw exception;
    ----
  }
catch (Datatype argument)
{  // statements that handles exception
  }
}
```

eg 1 → Exception Handling program →

```
# include <iostream>
  void main()
{ int f, s;
  cout << " Enter the first no.";
  cin >> f;
  cout << " Enter the second no. ";
  cin >> s;
    try
  { if (s != 0)
```

③
```
{ cout << "division =" << f/s << endl;
}
    else
    { throw (s);          // throw exception
    }
}
catch ( int e)
    { cout << " There is an exception division by" << y << endl;
    }
}
```

Throwing Mechanism →

The throw keyword is used to indicate that an exception has occurred. This is called throwing an exception. A throw specifies one operand that can be of any data type.

An exception can be thrown in any of following forms:

```
throw(exception);
throw exception;
throw;    // used for rethrowing an exception
```

When an exception is thrown it will be caught by the nearest catch block whose argument type match with the type of exception.

Catching Mechanism → Code for handling exceptions

is included in catch block. A catch block looks like a function definition and is of form:

```
catch ( DataType argument)
{ // statements for handling exception
}
```

The catch statement catches an exception whose data type matches with the data type of the argument of catch statement. When it is caught, the code in the catch block is executed.

After executing the catch handler, the control goes to the statement immediately following the catch block.

Multiple Catch Statements →

A program segment can throw more than one exception. In such cases, we can associate with more than one - catch blocks with a try block: Syntax →

```
catch ( DataType1 arg1)
{ // statements for handling exception
}
catch ( Datatype2 arg2)
{ // statements
}
```

(5)

```
catch (Datatype1 arg1)
{ // statements for handling exception
}
```

When an exception is thrown, the exception handlers are searched in order for an appropriate match. The first handler that yields a match is executed. After executing the handler, the control goes to the 1st statement after the last catch block for that try block i.e. all other handlers are bypassed. When no match is found, the program is terminated.

Catch all exceptions →

There may be some situations, where it may not be possible to anticipate the possible type of exceptions. It will not be possible to write separate catch handlers to catch them. The solution for this is to write a catch handler that can catch all exception instead of catching an exception of particular type.

This is achieved by defining catch statement using three ellipses as -

```
catch (...)
{ // statements
}
```

It may be good idea to use catch(...) as a default statement along other catch handlers that it can catch all those exceptions w/c are not handled explicitly.

eg

```
#include <iostream>
using namespace std;
void test (int x)
{ try
  { if (x == 0) throw x;      // int
    if (x == -1) throw 'x';   // char
    if (x == 1) throw 1.0;    // float
  }
  catch(...)                  // catch all
  { cout << "Caught an exception";
  }
}
int main()
{ cout << " Testing Catch ";
  test (-1);
  test (0);
  test(1);
  return 0;
}
```

O/P

Testing Catch
Caught an exception
" " "
" " "

Specifying exceptions → To restrict a fxn to throw only certain specified exceptions. This is achieved by adding a throw list clause to the fxn. definition. Syntax →

(1)

```
Datatype fxnName(arg-list) throw(exception list)
{ // fx. body
}
```

Throwing any other type of exception will cause abnormal program termination.

eg →

```
# include <iostream>
using namespace std;
void test (int x) throw(int, double)
{ if (x==0) throw 'x';   // char
  else
    if (x==1) throw x;   // int
  else
    if (x==-1) throw 1.0;   // double
  cout << " end of fx. body ";
}
int main()
{ try
  { cout << " Testing throw restrictions ";
    cout << " x==0 \n";
```

```
test(0);
cout << " x = = 1 ";
test(1);
cout << " x = = -1 ";
test(-1);
} ~~cout~~ catch (char c)
{ cout << " Caught a character ";
}
catch ( int m)
{ cout << " Caught an integer ";
}
catch ( double d)
{ cout << " Caught a double ";
}
cout << " End of try catch system ";
return 0;
}
```

O/P Testing throw restrictions

x = = 0

Caught a character

End of try catch system

Rethrowing an exception → A handler may decide to rethrow the exception caught without or partially processing it. In such situations, we can simply invoke throw without any argument as -

```
throw;
```

This causes the current exception to be thrown to the next enclosing try/catch sequence and is caught by a catch statement listed after this try block.

Unit-10

Templates

Generic programming → It is an approach where generic types are used as parameters in algorithms so that they work for a variety of suitable data types and data structures.

Templates → Templates are a way of making your classes more abstract by letting you define the behaviour of the class without actually knowing what datatype will be handled by the operations of a class.

A template can be used to create a family of classes or functions. eg. a class template for an array class would enable us to create arrays of various data types such as int, float array.

Two types →

1) Class Template 2) function Template

I. **function Template** → function template can be used to create a family of functions with different argument types. eg → Syntax

(2) template < class T1 , class T2 , --->

Datatype functionname (arguments of type T1, T2 ---)

```
{
   // function body
}
```

All fx. templates begin with a keyword template followed by a list of formal arguments to the fxn. template enclosed in angle brackets.

eg.

```
template < class T >
void swaptwovariables( T &a, T &b)
{  T temp;
   temp = a;
   a = b;
   b = temp;
}
```

Program →

```
# include <iostream>
# include <iomanip>
template < class T >
void swaptwo variables (T &a, T &b)
{  T temp;
   temp = a;
   a = b;
   b = temp;
}
void main( )
{  char ch1 = '*', ch2 = '+';
```

```
cout << " before swapping ";
cout << " first character variable" << ch1 << endl;
cout << " second character variable" << ch2 << endl;

    swaptwovariables ( ch1, ch2);
cout << " After swapping ";
cout << " first character variable" << ch1 << endl;
cout << " second character variable" << ch2 << endl;

  int intvar1 =10 , intvar2 = 20;

cout << " Before swapping";
cout << " first integer variable" << intvar1 << endl;
cout << " second integer variable" << intvar2 << endl;
  swaptwovariables ( intvar1, intvar2);
  cout << " After swapping ";
  cout << " first integer variable" << intvar1 << endl;
  cout << " second integer variable" << intvar2 << endl;

}
```

Class Templates →

Syntax → template < class T_1, class T_2 ---->
class classname
{ // class member specification with type T_1, T_2
}

All class templates begin with keyword template followed by a list of formal parameters to the class

(4) template enclosed in angle brackets where each formal parameter must be preceded by keyword class

eg, ~~class Item~~

~~{ void Data;~~

```
template <class T>
 class  Item
{  T  Data;
public:
 Item ( ): Data (T ())
   { }
 void  setdata ( T n~~ata~~)
   { Data = n;
     }
/*  T   getdata ( ) const
   { return  Data;
    }  */
 void  printdata ( )
 { cout << data;
 }
};
  int  main ( )
{  Item <int>  item1;
 item1 . setdata ( 120);
 item1 . printdata ( );
  Item <float>  item2;
// float  n  =  item2 . getdata ( );
 item2 . Setdata (150.2);
 item2 . printdata ( );
 return 0;
}
```

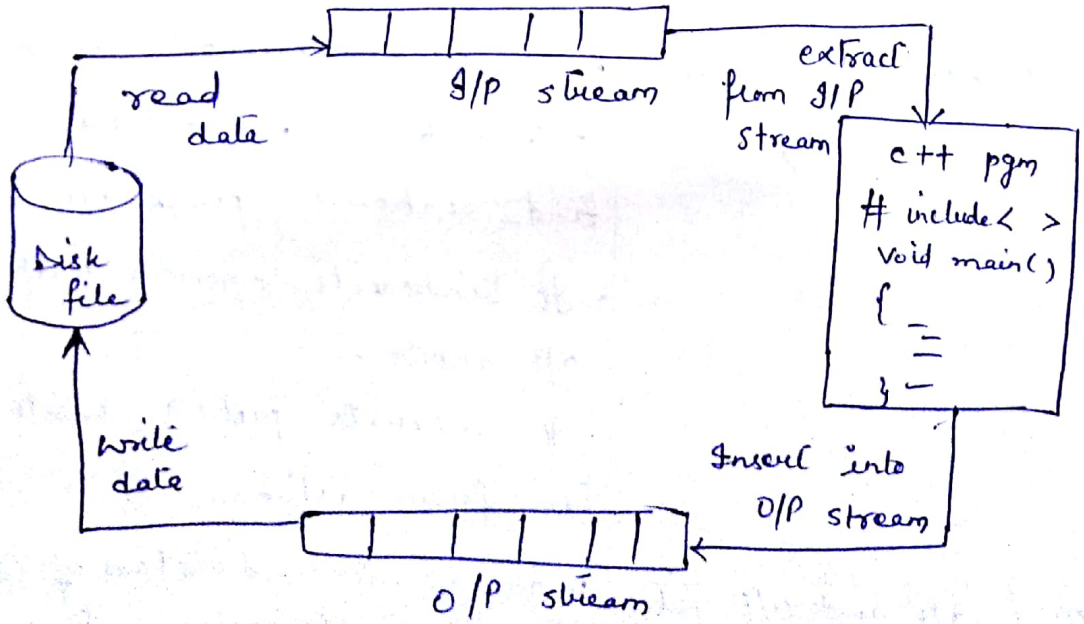
Unit-II (files)

File → A data file is a collection of records.
eg- file containing records of all students in a class.

Record → It is a collection of related data items.
Where data item refers to a single unit of values.
eg → data items are roll no., name, dob etc.

file organization refers to the way records are physically arranged on a storage device.

file streams → A stream is basically a sequence of bytes. It can either act as a source from which the program can extract input data or as a destination to which the program can send o/p data. The source stream that supplies data to the program is called input stream. The destination stream that receives data from the program is called output stream.



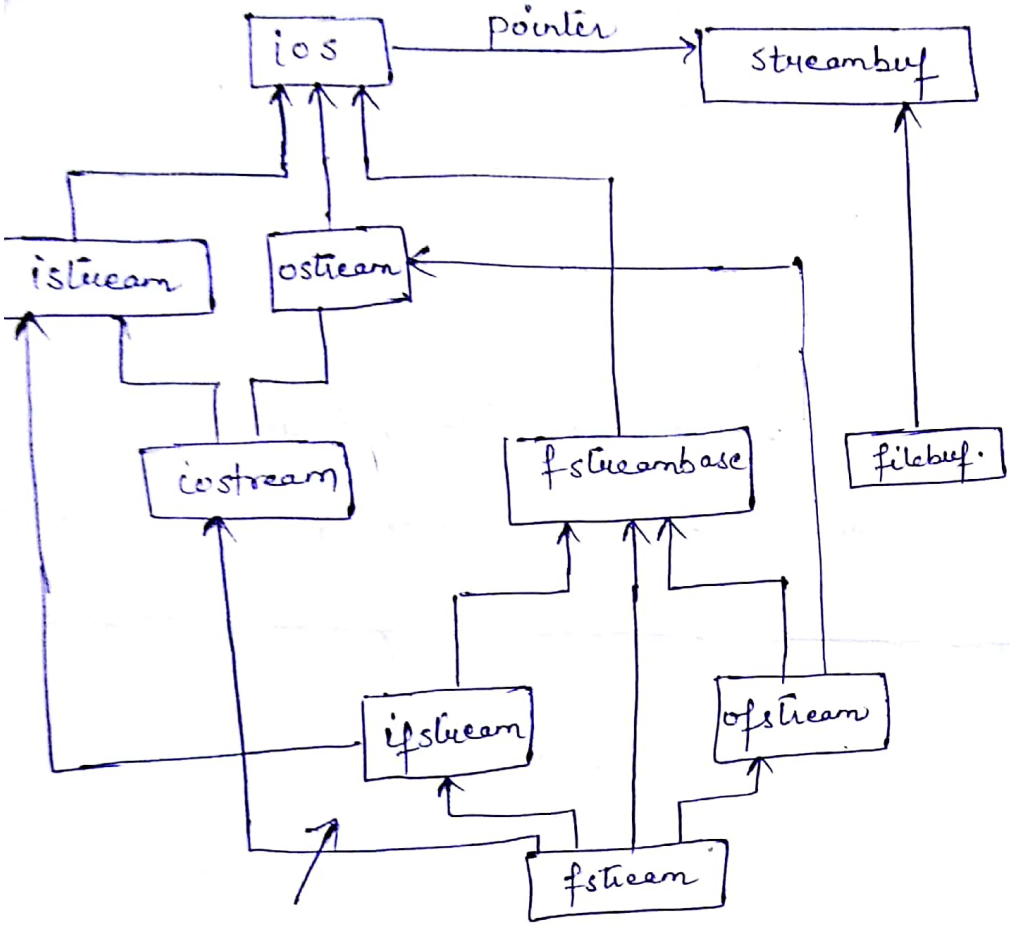
(2) Hierarchy of file stream classes →

C++ I/O subsystem contains a classes for handling data files.

Class Name	Description
1) filebuf	filebuf implements I/P and O/P to and from files.
2) fstreambase	• It is a base class for all file stream classes. It support operations common to file streams. • Contains open() and close() fxn.
3) ifstream (I/P file stream class)	• It is derived class of fstream base and istream classes. It inherits the properties of both classes. • It inherits open() with default input mode. • It inherits get(), getline(), read, seekg() fxns. from istream class.
4) ofstream (O/P file stream class)	• It is a derived class of fstreambase and ostream classes, and inherit properties of both. • It inherits open() with default O/P mode. • It inherits put(), write(), seekp() fxn. from istream class.
5) fstream (I/P and O/P file stream class)	• It is derived class of fstreambase, ifstream, ofstream, iostream classes.

It supports simultaneous input and O/P operations.

Hierarchy of file stream classes



file stream class hierarchy

Steps in processing a file →

(1) **Naming a file** → I^st step in file processing is to select an appropriate name. Name of file indicate the type of its contents. The rules for naming a file may be different for diff. operating systems.

(4)

2) Opening a file
3) Reading from or writing onto a file
4) Closing the file.

Opening a file → for opening a file, first we to create a file stream and then link it with; A file stream can be created using class ifstream, ofstream, fstream that are defined in fstream.h header file. 2 (two) ways –

1) Using constructor of appropriate file stream class.
2) Using open() member fxn. of appropriate file stream class.

1) **Opening a file using constructor** →

Constructor is used to initialize an object while it is being created. filename is used as an argument for the constructor, which is then used to initialize the file stream object.

Syntax for ifstream class is →

```
ifstream( const char *path, int mode = ios::in,
          int prot = filebuf::openprot);
```

where, path is filename with path info. mode is file

ing mode, by default value is 'in' for I/P mode. (5) (8)
It is access specifier, default argument with default value 'openprot' for read & write access permissions.

for ofstream class →

ofstream(const char *path, int mode = ios::out,
int prot = filebuf::openprot);

fstream (const char *path, int mode = ios::in |
ios::out, int prot = filebuf::openprot);

file mode parameters are -

binary → Open file in binary mode. by default, file is opened in text mode.

in → open file for read only

out → open file for write only

trunc → Delete the contents of the file if it exists.

app → Append to end of file

ate → move the file pointer to the end of file on opening.

Access permission

S_IREAD → Read permission

S_IWRITE → Write "

S_IEXEC → Execute permission

eg. ofstream outfile("rating.dat", ios::binary|

(6) Open in binary mode for o/p with read or write permissions.

(2) Opening a file using open() →

Prototype of open() →

void open (const char *path, int mode, int prot = filebuf :: openprot)

eg- ifstream infile;
infile.open ("score.dat", ios::in)

Closing a file → When a program has finished with reading / writing of a file, it must be closed by using close() of fstreambase class.

Syntax → void close();

eg samplefile.close();

Reading & Writing of files →

(1) Reading & Writing of character data → character data is read & written using character input/output member function of ifstream and ofstream classes.

I. Writing character data → Write/create a file using

2 Character O/P fun. put() of ofstream class. The program reads data from the keyboard & writes to a file.

```
#include <iostream>
#include <fstream>
void main()
{ char ch;
  char filename[12];
cout << "Enter name of file";
cin >> filename;
ofstream outfile(filename);
if (outfile.fail())
{ cout << "\n unable to
open file" << filename << endl;
return;
}
cout << " Enter some text & terminate"
<< "^z followed by Enter key";
while ( ! cin.eof())
{ cin.get(ch);
  outfile.put(ch);
}
outfile.close();
}
```

O/P Enter name of file : file1.dat
Enter some text & terminate by ^z
Enter key
It is better to keep my word soft and sweet.

(2) Reading character Data →

The program is used to read a file using get() of ifstream class. It reads data from the file and displays on monitor.

```
#include <iostream>
#include <fstream>
void main()
{ char ch;
  char filename[12];
cout << "Enter name of file to rea
cin >> filename;
ifstream infile (filename);
if (infile.fail()) {
```

(8)

```
cout << "\n unable to oper file" << filename <<
    return ;
}
cout << "contents of file" << filename << "\n";
  while ( ! infile . eof())
  { infile . get (ch);
     cout << ch;
  }
  infile . close();
}
```

O/P

Enter name of file to read : file1.dat

Contents of file :

It is better to keep my word sweet & soft

(2) Reading & Writing of String Data →

(1) Writing String Data → uses write().

```
# include <iostream>
# include <string>
# include < fstream>
void main()
{
  char filename[12], stname[81];
  int stlength;
cout << "Enter name of file";
  cin >> filename;
  ofstream outfile ( filename);
  if ( outfile . fail())
{
cout << "Unable to oper file" << filename << endl;
return;
}
cout << "Enter some strings and terminate by null string << endl;
while (1)
{ cin . getline(stname, 80);
# stlength = strlen(stname);
```

```
if (stulength == 0)
  break;
outfile.write(stuname, stulength);
}
outfile.close();
}
```

O/P

Enter name of file:
file2.dat
Enter some strings and terminate by null string.

The winner never quits ↵
Eyes are index of your heart ↵

Reading string data → It uses getline() of ifstream class.

```
#include <iostream>
#include <fstream>
#include <string>
void main()
{ char filename[12],
  stuname[80];
cout << "Enter name of file";
cin >> filename;
ifstream infile (filename);
if (infile.fail())
{
cout << "\n unable to oper file"
  << filename << endl;
  return;
}
cout << "\n contents of file"
  << filename << endl;
while ( ! infile.eof())
{
infile.getline (stuname, 80);
cout.write(stuname, strlen(
        stuname));
```

```
    cout << endl;
}
    infile.close();
}
```

```
Contents of file: fa
The winner never q
Eyes are index of w
    heart.
```

Error Handling during file operations →

There are some situations may arise while handling the files:

1) Attempting to open a non-existence file in read mode.
2) Attempting to open a file in write mode that was created with read only access permission.
3) Attempting to open a file with invalid filename.
4) Sufficient disk space is not available when writing to a file.
5) Attempting Invalid operations.

To detect such situations there are some fns.

1) eof() → Return non zero (true) value if EOF encountered while reading a file otherwise 0.
2) fail() → Returns non-zero if oper, read, write has failed, otherwise zero
3) bad() → Returns non-zero, if invalid operation is attempted else zero.

ferror() → Returns non-zero if everything is fine, otherwise zero.

clearerr() → Clear the current error state so the further operations can be attempted.

Accessing Records Randomly →

Writing and reading operations on file are performed in sequential order. But it is also possible to access a particular data item that may be in the middle of the file. This way of accessing a file is called direct accessing or random accessing. The only restriction for this ~~addressing~~ accessing is that all data items must be of same size. This is performed by use of pointers.

file pointer →

A file pointer is a pointer to a particular byte in a file.

→ Every time, when we write to a file, the file pointer moves to the end of data item written so that writing can continue from that point.

→ When a file is closed, subsequently opened for reading, a file pointer is set to beginning of file.

→ If the file is opened in append mode, then file

(12) pointer will be positioned at the end of existing file, so that new data items can be from there onwards.

The C++ language file mgmt system associates file pointers with files. These pointers are called get pointer (I/P pointer) and put pointer (O/P pointer)

These pointers have following purpose:

1) Get pointer specifies a location in input file from where the current read operation is initiated.
2) Put pointer specifies a location in O/P file from where current write operation is initiated.

There are some situations where these file pointers are to be set at desired position in a file so that read or write operation can take place. The file stream classes contain member fxns that allow moving these file pointers within a file.

fxns	Class in which defined	Action performed
1) seekg()	ifstream class	→ Moves get file pointer to specified position.
2) seekp()	ofstream class	→ Moves put file pointer to specified position.
3) tellg()	ifstream class	→ returns the position of get pointer
4) tellp()	ofstream class	→ returns the position of put pointer

updation → The most usual way of updating a sequential file is to read data items from one file, update the required data items & write onto a new file. This is shown by an example a file containing student's marks in subjects. It is decided that those students who had secured less than 50 marks in english be given 5 grace marks.

```
# include <iostream>
# include <fstream>
  class studenttype{
   private :
    int rollno;
      char firstname[20];
      float engmarks;
      float mathmarks;
      float phymarks;
    public:
    Studenttype ( ) { }
    void read( )
  { cout << " Enter roll no";
    cin >> rollno;
    cout << " Enter first name";
    cin >> firstname;
    cout << " Enter marks in english ";
    cin >> engmarks;
    cout << " Enter marks in maths";
    cin >> mathmarks;
    cout << " Enter marks in
      physics";
    cin>> phymarks;
    }
    void print ( )
    { cout <<"roll no" << rollno;
    cout <<" first name"<< firstname
                        ;
    cout<<" marks in eng" <<
        engmarks;
    cout <<"marks in maths"<<
        mathmarks;
    cout <<" marks in physics "<<
      phymarks ;
                }
```

(14)

```
void update()
{ if (engmarks < 50)
    engmarks += 5;
}
};  void main()
{
    studentType student;
    ifstream infile ("oldfile.doc", ios::binary | ios::in);
    if (infile.fail())
    { cout << "\n unable to open file" << endl;
      return 0;
    }
    ofstream outfile ("newfile.doc", ios::binary | ios::out);
    if (outfile.fail())
  { cout << "\n unable to open" << endl;
    return 0;
  }
    while (!infile.eof())
    { infile.read((char *) &student, sizeof(student));
      if (infile.bad())
      { cout << "\n error while reading old file";
        return 0;
      }
    student.update();
    outfile.write((char *) &student, sizeof(student));
      if (infile.bad())
        {
```

```
cout << "\n error while writing to file newfile";
return 0;
}
}
        infile.close();
        outfile.close();
}
```

(15) (8)