OBJECT ORIENTED PROGRAM AND STRUCTURE

4-8-22

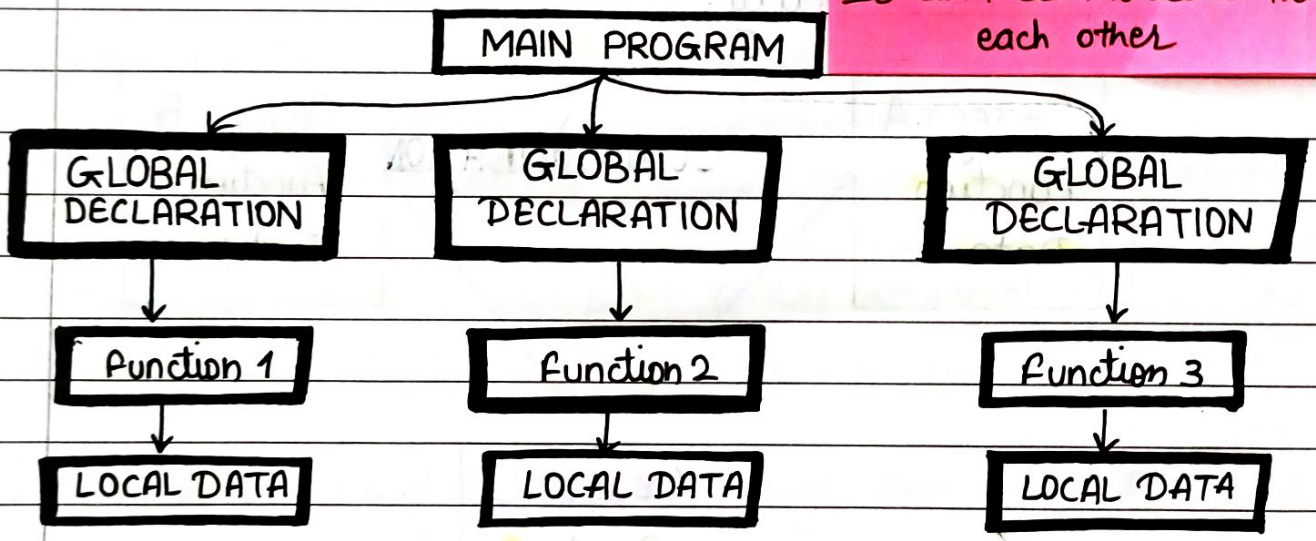
Programming Paradigm.

Methods to solve any problem using programming language is called programming paradigm.

→ PROCEDURAL PROGRAMMING PARADIGM } indicates C language.

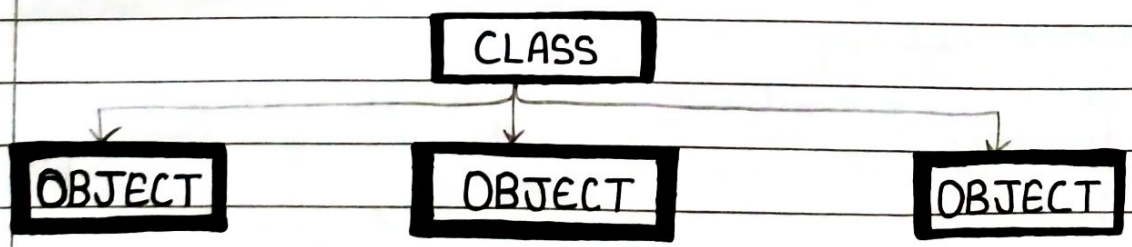
It consist of series of computational steps to be carried out. The procedures are : functions etc. It follows step by step approach {Top to bottom} in order to break down a task.

It can't communicate through each other



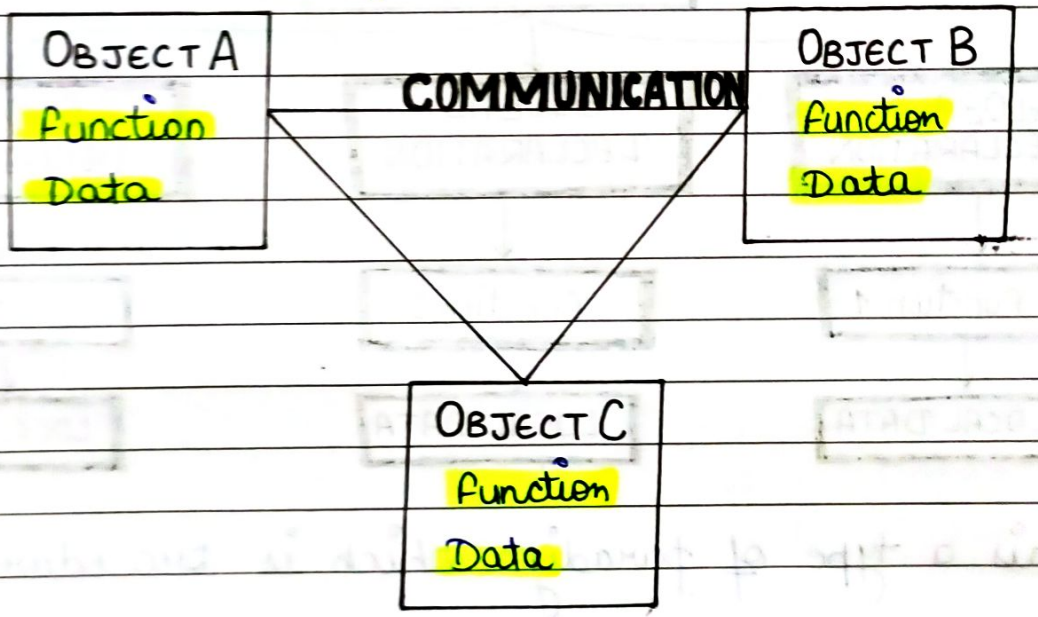
This a type of paradigm which is procedural {C lang}.

→ OBJECT ORIENTED PROGRAMMING PRADIGM } indicates C++ language



Just like C which has global declaration then function and then local data as a part of programming it too has a main part called class and then object which includes characteristics of class as well as has its own.

It includes 2 words : object and oriented
It has : class, object, data security, data abstraction, data hiding, polymorphism, function overloading, operators overloading, code repeatation, code reversability.



__/__/____

DIFFERENCE BETWEEN OBJECT ORIENTED AND PROCEDURAL PROGRAMMING.

	PROCEDURAL PROGRAMMING	OBJECT ORIENTED PROGRAMMING
DEFINITION:	Programming model which is derived from structured programming, based upon the concept of calling procedure.	Programming mode which is based upon the concepts of objects. Objects contain data in form of attributes and code.
DIVISION:	The program is divided into small parts called functions.	The programs are divided into small parts called object.
APPROACH:	It follows top-down approach.	It follows bottom-up approach.
ACCESS SPECIFIER:	There is no access specifier	It has access specifiers like public, private, protected.
ADDITION OF DATA	: Adding new data and functions is not easy.	Adding new data and functions is easy.
SECURITY OR DATA HIDING	: Since, it doesn't have any proper way of hiding data so it is less secure.	It provides data hiding so it is more secure.
OVERLOADING:	Overloading is not possible Ex → Sum if we name diff func^n as SUM it will show error	overloading is possible. Ex → sum (int a) Sum (float a) Sum (int b, int c) we can use diff. functions same name.

Using of same name for diff functions, called overloading.

what is inheritance ?
what is data abstraction ?

__/__/____

	PROCEDURAL PROGRAMMING	OBJECT ORIENTED PROGRAMMING.
INHERITANCE:	There is no concept of data hiding and inheritance	There is a concept of data hiding and inheritance.
IMPORTANCE OF FUNCTION OVER DATA.	: Function is more important than data	Data is more important than function.
BASES OF PROGRAMMING	: Programming is based on unreal world.	Programming is based on real world.
DESIGNING SIZE	: Programming is used for designing medium-sized programs.	Programming used for designing large and complex programs.
CONCEPT :	It uses the concept of procedure abstraction	It uses the concept of data abstraction.
CODE REUSABILITY	: Code reusability absent in procedural programming Example : C, Pascal, FORTRAN, Basic etc	Code reusability present in object oriented programming Example : C++, java, python etc.

__/__/____

To write C++ program →

step1: we start with

In C : we use
include <stdio.h>

```
# include <iostream>
```

used for managing input and output streams.

step2: declare func^n called main with the return type of int. main() is entry point of our program.

just like C

```
int main ()
```

step3: Starts curly brackets ({).

Now, this line has 3 parts as:

1st std :: cout

which identifies the standard console output devices.

In C, we use printf and scanf, here we use cin>>, cout

2nd <<

the insertion, which indicates that what follows is inserted into std :: cout.

3rd we have sentence within quotes that we'd like printed on the screen.

step4: End of curly braces (}).

__/__/____

CODES IN C++

Ques1 → Write a program for the addition of 2 no.

Soln →

```
#include <iostream>
int main()
{
    int a,b;
    int sum;
std::cout << "Enter 2 numbers a and b :";
std::cin >> a >> b;
    sum = a+b;
std::cout << "sum =" << sum;
return 0;
}
```

Output →

Enter 2 numbers a and b : 12
13
sum = 25

__/__/____

Ques 2) → Write a program to find area of circle in C++.

Soln →

```
# include <iostream>
int main ( ) {
float r , area ;
std:: cout << "Enter the value of r:" ;
std:: cin >> r ;
area = π r*r;
std:: cout << "area=" << area;
return 0;
}
```

Enter the value of r : 2

area = 12.57

Using "if-else" statement

Que 3) → Write a program if a no. is -ve, +ve or zero.

Soln →

to use this step we follow →

```
if (test expression)
{
statement block 1;
}
else (test expression)
{
statement block 2;
}
statement x;
```

```
# include <iostream>
int main ( ) {
int a, +ve , -ve , zero ;
std:: cout << "Enter any digit a: " ;
std:: cin >> a ;
if (a > 0) {
std:: cout << " +ve " ; }
else if (a < 0) {
std:: cout << " -ve " ; }
else
std:: cout << " zero " ; }
return 0 ;
}
```

output :

enter any digit a: -6

-ve

__/__/____

Array

collection of similar data type.

declaration of array ⇒

data type array name [size];

int array1 [10];

initialization ⇒

int array1 [10] = {1,2,3,4,5,6...10}

int array1 [ ] = {1,2,3,4,5...10}

1	2	3	4	5	6	7	8	9	10

int array1 [10] = {1,2,3}

1	2	3	0	0	0	0	0	0	0

Accessing elements of array →
with the help of index.

0	1	2	3	4

array index → 0 1 2 3 4

array elements → array[0] array1[1] array1[2] array1[3] array1[4]

No index out of bound checking.

__/__/____

```
int main ()
{
    int arr [1];
    arr [1] = {2}
cout << arr [-1];   } we can't use -ve hence out of bound.
cout << arr [2];    } declared value is 1 so we can't use 2 hence out of bound.
```

if :→

int array1 [2] = {10, 11, 12, 13}

→ c
→ compilation but shows warning.

→ c++
→ no compilation
→ shows error

[too many initialization for int array1[2] error can be like this.]

1) Write a program to print value from user and show the same output using array.

Sol →

```
# include <iostream>
int main ()
{
int array1 [5];
std:: cout << "Enter 5 elements: ";
for (int i = 0; i < 5; i++) {
std:: cin >> array1[i]; }
```

__/__/____

```
for (int j=0; j<5; j++){
    cout << array1[j];
}
return 0;
}
```

to print address of array1[0].

```
cout << &array1[0];
```

but we need to print address of every array we need to write it in for loop.

2) Write a program to print the sum of array.

Soln:

```
#include <iostream>
int main()
{
int array[5];
int sum=0;
std::cout << "Enter 5 elements";
for (int i=0; i<5; i++) {
std::cin >> array[i]; }
for (int j=0; j<5; j++) {
    sum = sum+array[j];
}
std::cout << sum;
return 0;
}
```

3) Write a program to find the average.

Soln →

```
# include <iostream>
int main ()
{
int arr[5]; int x=0;
std::cout<<"Enter any 5 elements:";
for (int i=0; i<=5; i++)
{
std::cin>>arr[i];
}
for (int j=0; j<=5; j++)
{
x=[x+arr[j]];
}
std::cout<<"x/5";
return 0;
}
```

__/__/____

#

STRING

collection of character OR Array of character.

Declaration →

Data Type name [ ]

Char str1 [4];

Inbuilt class : string

C++ provides 2 types of string representation :
- C style character string
- strings that are object of string class.

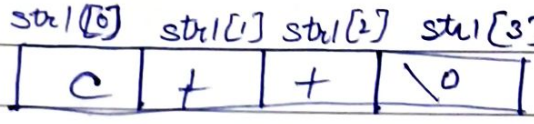
initialization ⇒

char str1 [4] = "C++";

OR

Char str1 [4] = { 'C', '+', '+', '\0' };

Memory representation



__/__/____

1) Write a program to print string through user.

Soln →

```
#include <iostream>
int main()
{
    char str.[ ];
std:: cout<<"Enter your name";
    Std:: cin >> str.[ ];
std:: cout<<" your name is :"<< str.[ ];
    return 0;
}
```

no need to write size

output : Enter your name :
Akriti Katoch
Your name is :
Akriti

to print even after space we use cin.get instead of cin
or
cin.getline(name, size);

cin >> str. } instead cin.get(str., [ ]);

Same program using string class ↓.

```
#include
int main() {
    string str.;
std:: cout << "Enter your Name";
std:: getline (cin, str.);
std:: cout << " your name is :";
    return 0;
}
```

small → [string str.;] no need to add size

__/__/____

String Functions :→

strcpy (S1, S2); ← this string will be copied in S1

strcat (S1, S2); } joines
concatination string (S1, S2)

strlen (S1); } length

strcmp (S1, S2); } comparing both string
returns 0 when ?

strchr (S1, Ch); } points on the character we mentioned using (Ch) in the string(S1)
if similar words are there than blinks on 1st positioned letter.

strstr (S1, S2); } blinks on the string present in S1.
S1 = Hello World;
S2 = World

S1 = Hello
S2 = World
S3 = Declare only

__/__/____

1) Write a program 2 strings as Hello, world and declare the empty string using copy, compare and length i.e string function

soln →

```
#include <iostream>
#include <cstring>
int main()
{   int length;
    string S1 = "HELLO";
    string S2 = "WORLD";
    string S3;
        S3 = S1;        ] strcpy(S3, S1);
    std::cout << S3;
        S3 = S3 + S1;   ] strcat(S3, S1);
    std::cout << S3;
std::cout << length = S3.size(); strlen(S3);
    return 0;
}
```

in C ↓

```
char S1[10] = "HELLO";
char S2[10] = "World";
char S3[0];
```

__/__/__

POINTERS

type of variable, holds the address of other variable

Declaration

data type *variable name ;

```
# include <iostream>
main();
int a = 10;
int *ptr;
ptr = &a;
Std:: cout <<ptr; address
Std:: cout <<*ptr; value
Std:: cout <<"a";
```

__/__/____

1) Write a program of swaping 2 variable using pointer.

Sol^n.

```
#include <iostream>
int main()
{
int *a, *b;
int x=10; y=20, c;
std::cout<<"Before swaping "<< x <<y <<endl;
a=&x;
b=&y;
c=*b;
*b=*a;
*a=c;
std::cout<<"After swaping "<<x<<y <<endl;
return 0;
}
```

Recursion program →

```
#include <iostream>
int main ()
{
int i;
```

```
#include <iostream>
int factorial (int n);
int main ()
{
int n;
std:: cout << "Enter a positive integer : ";
std:: cin >> n;
std:: cout << "Factorial of " << n << " = " << factorial
(n);
return 0;
}
int factorial (int n) {
if (n > 1)
return n * factorial (n - 1);
else
return 1;
}
```

__/__/____

CLASS :→ It is a user defined data type which holds its own data members and member function.
Data Members are the data variables and Member Functions are the functions used to manuplate the variable.

Example → 1| furniture = Class
chair, table etc = object.

2| Blueprint of a building = Class
actual building = object.

3| consider a class of cars have different names and brands but have same properties like speed, 4 wheels etc.

- class = cars Data Member = speed, 4 wheels
Member function = applying breaks, inc/dec speed.

```
Class: useridefined name
{
  Acess specifier : //can be public, protected
                    or private.
  data member ; // variable
  member function ( ) ; // function.
};
```

__/__/____

```
Class Student
{
  public :
    string name ;
      void display();
};
```

OBJECT ⇒ it is an instance of a class when a class is defined no ~~member~~ memory is allocated but when object is created memory is allocated to class.

```
class userdefined name
{
:
:
};
class name variable name;
```

Example → only for public.

```
#include <iostream>
class student
{
public ;
  string name ;
  void display ( ) ;
  {
std : : cout << name ;
  }
};
void main ( )
```

```
{
    student obj. ;
    obj. name = "ABC" ;
    obj. display ( ) ; // function calling
}
```

Accessing Data Member and ~~Data~~ Member func^{n} of class:

We a data of a class is accessed using dot operator.

Syntax ↓.

class object. data member

class object . member func^{n}.

Declaring ~~Accessing~~ Member func^{n}

we declare ~~access~~ member func^{n} by 2 ways →

1) inside class.
2) Outside class.

inside/outside =

```
class student
{
public :
    String name ;
    int id ;
    void display string ( ) ;
    {
        cout << name ;
    }
    void displayid ( ) ;
};
```

inside

```
void student :: display id ()
{
  cout << id ;
} main ()
  { student obj ;
obj. name = "ABC" ;
obj. id = 12 ;
obj. display string () ; // func calling
obj. display id () ;
```

outside

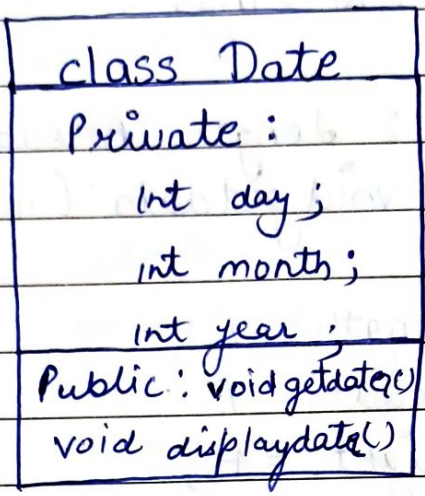
1) Write a program for the addition of 2 no:s.

```
# include <iostream>
  class add
{
  public :
   int a, b, sum ;
   void display sum ()
  { sum = a+b ;
  cout << " sum. " ;
  }
  }; main ()
   { add obj ;
   obj. a = 5
   obj. b = 6
   obj. sum = a+b ;
  obj. display sum ( ) ;
  obj. display (5,6)
  }
```

__/__/____

1) Write a program to demostrate how to define both data member and member func^{n}. with in the scope of class.

=>




```
Class Date
{
private : int day, mont, year;
public : void getdata (int d1, int m1, int y1)
   {
     day = d1 ;
     month = m1 ;
     year = y1 ;
   }
Void display data ( )
{
 cout << "Today date is" << day << month << year;
}
};
  Void main ( )
Date obj.
int d1 = 10, y1 = 1995 m1 = 2;
obj. getdata (d1, m1, y1);
   obj display ( );
```

2) Class Rectangle, parameters = LBH and find area and volume.

```
Class Rectangle
{
private : length, Breadth, Height ;
public : void getdata (int L, int B, int H)
  {
    length = L;
    Breadth = B;
    Height = H;
  }
  Void display data (int A, int V)
  {  int A = L*B ;  int V = L*B*H
Std:: cout << "Area of rectangle is :" << A;
  }
  };
  void main ()
  Rectangle obj;
int L = 2;
int B = 4;
int H = 1;
obj. getdata (A
```

__/__/____

```
class Rectangle
{
  private:
     int length, breadth, height;
  public:
       void area (int L, int B)
        {
        length=L;
        breadth=b;
  cout << "Area is :" << length * Breadth ; }
  void volume (int L, int b, int h) {
        length =L;
          breadth = B;
          height = H;
  cout << "volume is : " << length * breadth * height;
     }
     };
  void main ()
     Rectangle obj ;
       obj. area (10, 4);
         obj. volume (10, 4, 6);
```

1) no inheritance in private
2) more secure private

Access Specifier.

private, protected, public

1) Public ⇒ using it data member, member func^n has access throughout the class.

The members which are declared in public func^n can be accessed by any func^n out of the class.

The Public implementation operations are also called as member func^n or methods or interfaces ~~to out~~ through out of the class.

2) Private ⇒ in a private section a member data can only be ~~asse~~ accessed and friends of this class. Private data member is not accessible to the outside of the class

3) Protected ⇒ The members which are declared in the protected section can only be accessed by the member func^n and friends of this class. Also these func^n can be accessed by member func^n and friends derived from this class. It is not accessible to the outside world.

Nested Classes / classes within classes.

enclosing class → class A {

nested class ←

public:

class B {

same access specifier

};

};

__/__/____

1) A nested class is a class i.e declared in another class.
2) It is also a member variable of the enclosing class and has the same access rights as the other members
3) The member func^n of the enclosing class have no special access to member of nested class.

Syntax:

```
class outer classname
{
  Private : data member;
  Protected : data member, member func$^n$;
  Public : member func$^n$;
    class innerclassname
    {
    Private : data member;
    Protected : data member, member func$^n$;
    Public : member func$^n$;
    };
};
```

object of innerclass under outerclass → outerclass :: innerclassobj;

space obj] add space

Example →

```
classA {
Public:
  classB {
Private: int number;
Public: void getdata (int n);
  {
```

```
        number = n;
    }
    void display data ( ) {
        cout << number;
    }
};
int main ( )
{
A : : B obj ;
obj. getdata (8);
obj. display data ( );
return 0;
}
```

Local Class

A class declared inside the func^n becomes local to that func^n and is called local class in C++.

```
#
using namespace std;
void func^n ( ){
local class <— [ class classname {
    };
}
int main ( )
{
  fun ( );
  return 0;
}
```

__/__/____

⇒ A local class name can only be used inside the func^{n} and not outside it.

#

member func^{n} / Data member → methods

```
using namespace std;
void fun() {
    class local class :
    public:
        void method();
    };
    A obj;
    obj.method();
}
int main()
{
    A obj; // error
    obj.method(); // error
    fun();
}
```

⇒ All the methods of local class must be defined inside the class only

⇒ A local class can't use static data member

Arrays and Classes.

__/__/____

→ Array of ~~vctor~~ objects, store objects
→ An array of class type is known as an array of objects
Example: storing more than 1 student data

Syntax

```
class name  object [size ];
```

Program

```
class student
{
 private : char name[20];
           int rollno.;
 Public : void getdata ();
          Void displaydata ();
};
void student :: getdata ()
{
cout << "Enter name";
cin >> "name";
cout << " enter rollno.";
}
void student :: displaydata ()
{
cout << name << ~~display~~ rollno.;
int main ()
{
student obj [30];
int i, n;
cout << " Enter no. of students ";
cin >> n;
```

member name

```
for (i=0; i<n; i++)
{
  cout obj.[i].getdata ();
  }
for (i=0; i<n; i++)
  {
    obj.[i].displaydata ();
  }
}
```

Pointers and Classes. → in c++ pointers are variable they indicator which indicate address. performance of code increases & length of code is reduce.

Dynamic memory allocation.

```
class Example
{
private : data member ;
public : member f^n ()
  =
};
Example obj. ;
Example *ptr ;
```

Here ptr is a pointer variable that holds the address of class object.

The pointer to an object class variable will be accessed and processed in one of the following ways

calling func^n

```
(*obj.name).member name = value ;
```

syntax of calling func^n

__/__/____

Another Method :→

objname → member name.

↗ indirection operator i.e * operator

1) WAP to assign some values to the member of class, object using pointer structure operator.

student name
Roll. no.
Department and age.

⇒

```
#include<iostream>
using namespace std;
class student
{
Private : char name [100];
          int Roll no ;
          char dept.name [100];
          int age ;
Public : void getdata ( );
         void displaydata ( );
```

Samples

__/__/____

Static Class Members.

Static data member

static member func^{n}()

declaration →	Member func^{n}().
static datatype variable ;	static void sum ( ) ;
static int a ;	

→ STATIC DATA MEMBER.

1) Whenever we declare a datamember as a static either inside or outside of a class is called static data members

2) Static data members are data objects that are common to all the objects of a class.

3) They exist only once in all objects of this class.

4) It is always initialized with zero b/c its default value is 0.

5) static data members can be any one of the groups "Public, Private and Protected".

6) Static data member should be created and initialized before main func^{n} control blog begins

declaration of static variable

```
Class Example
{
Public : int x;
         static int y;
};
Example :: int y ;   → definition
int main ()
{
```

__/__/____

Example E1;
E1.x = 10
E1.y = 30

example E2;
E2.x = 20
E2.y = 40

```
cout << E1.x << endl;
cout << E2.x << endl;
cout << E1.y << endl;
cout << E2.y << endl;
```

Result →

x = 10
x = 20
y = 40

} y has only one memory space so it erased 1st output of 30 to store next output hence 2nd output becomes the overall result.

E1.x & E2.x has different memory space but E1.y & E2.y have only one memory space.

Aim
→ to demonstrate how an automatic initialization of a static member is carried out and the contents of the variables is displayed.

⟹

```
Class Object
{
public: int x;
        static int y;
        void display ();
};
object :: int y;
void object :: display ()
```

```
{
  cout << y;
}
int main()
{
  example object E1;
          E1.display()
  }
```

STATIC MEMBER FUNC^{N}.

1) The keyword static is used for member func^{n} to make member func^{n} static.

2) It access only static data member.

3) The static member func^{n} can't be virtual func^{n}.

4) A static or non-static member func^{n} can't have same name & same argument type.

5) It can't declare with the keyword 'const'.

```
class example
{Public : int x=10
 static int y;
Public : void display x()
 {cout<<x; }
 static void display y ()
 {
  cout << y $; }
  3;
int example :: y;
```

```
int main()
{ Example E1;
  E1.x = 10
  E1.displayx()
  E1.y = 20
  E1.displayy()
Example E2;
  E2.x =40
  E2.displayx()   E2.y =30
                  E2.displayy() }
```

6) static member func^{n} can be called directly using the class name and the scope resolution operator

Constructur

is a special method that is automatically invoked at the time of object creation.

[inside the class]

```
class classname
{
  classname ()
  {
    any task ;
  }
};
```

{outside the class}

```
class classname
{
  classname ();
};
classname :: classname ()
{
  code/any task ;
}
```

__/__/____

PROPERTIES / RULE OF CONSTRUCTOR

→ It has same name as the class itself.
→ It don't have return type.
→ It must be declared in public section.
→ If we donot specify constructor, C++ compiler generates a default constructor for objects.
→ It may not be static, and may not be virtual.

```
class construct
{
Public:
    construct()
    {
cout<<" constructor is calling ";
    }
};
int main()
{
construct obj, obj2, obj3; //3 times result
return 0;
}
```

in simple func^n while calling we have to use dot operator but here we arenot using any such operator.

C++ code to print no. of objects created using static variable. OR. A program to check how many instances of class of object are created using static member func^n.

```
class construct
{ static int count;
Public:
    construct()
    {
    count++;
    }
```

__/__/____

```
static void display()
  {
    cout << count;
  }
};
int construct :: count;
  int main()
  {
  construct obj1, obj2;
  construct :: display();
  }
```

Types of Constructor:

- Default constructor
  is the constructor which doesn't take any argument.

To print values of name and rollnumber of a student

```
class student
{
Public
  char name[10];
  int rollno;
```

```
class student
{
char name[ ];
int rollno. ;
public :
        student ( )
    { cin >> name ;
      cin >> rollno. ;
    }
      void display ( )
    { cout << name << rollno. ;
    }
};
```

- Parameterized Constructor
  in which we can pass parameters

```
class construct
{
   int a, b;
Public : construct ( int a1, int b1)
        {
            a = a1 ;
            b = b1 ; }
void display ( )
 {
 cout << a << b ;
 }
};
int main ( )
{ construct obj (10, 20) ;
  obj. display ( );
}
```

__/__/____

Explicit constructor.

```
#include <iostream>
using namespace std;
class abc
{
    int a1, int b1;
Public: abc (int a, int b)
    {
    a1 = a;
    b1 = b;
    }
void display ()
};
main ()
abc obj (10, 15);        implicit constructor
abc obj1 = abc (20, 30); explicit constructor
```

Copy Constructor

→ It is a member func" that initialize an object using another object of same class. It always used when the compiler has to create temporary object of that class. object.

→ It is used to initialize the members of newly created object. by copying the members of already existing object

& operator.

__/__/____

syntax →

```
class name (class name & old object)
```

OR

```
class name (const class name & old object)
```

```
class fibo
fibo :: fibo ()
{
 f0 = 0;
 f1 = 1;
 fib = f0 + f1
}
fibo ( fibo & obj)
{
 f0 = obj.f0;
 f1 = obj.f1;
 fib = obj.fib;
}
fibo obj;
```

→ The process of initializing members of an object through a copy constructor is known as copy initialization.

→ It is also known as memberwise initialization b/c the copy constructor initialize one object with existing object.

```
class construct
{
    int a1, int b1;
Public : construct (int a, int b);
    {
        a1 = a;
    } b1 = b;
```

copy constructor

```
construct (const construct & obj)
{
    a1 = obj.a1;
    b1 = obj.b1;
}
~~construct obj~~; void display ()
{
cout<<a1<<b1;
}
};
int main ()
{ construct obj (10,15); //para invok
  construct obj1(10,15); //copy invok
  obj.display ()
  obj1.display ()
}
```

HW.

1) write a program to generate fibonacii series using copy constructor where copy constructor is defined out of the class declaration using scope resolution operator.

__/__/____

Overloading Constructor.

→ the overloading constructor is a concept in oops in which the same constructor name is called with diff. arguments.

→ Depending on the type of argument the constructor will be invoked automatically by the compiler to initialize the object.

```
class construct
{
public: construct ();
        construct (int a);
        construct (float fa);
        construct (int a, float fa);
};
construct :: construct ()
{
cout <<"default";
}
construct :: construct (int a).
{
cout <<a;
}
construct :: construct (float fa)
{
cout <<fa;
}
construct :: construct (int a, float fa)
{
cout <<a<<fa;
}
main ()
```

NESTED CLASS

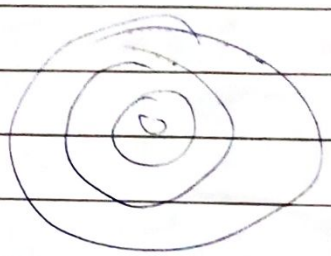
__/__/____

→ Constructor in nested class

1) The nested class is the technique in which a class is declared as a member of another class.

2) A class within a class is called a nested class.

3) The ::{scope resolution operator} is used to identify the outer and inner class object and constructor member func^{n}.



```
# include <iostream>
using namespace std;
class A
{
 public : A( ) ;
          class x {
          Public : x ( );
          };
};
A : : A ( )
  {
   cout << "A constructor";
  }
A::x::x( )
  {
   cout << " x constructor";
  }
Int main( )
{
  A obj ;
  A::x obj1;
}
```

output of outer class 1st then innermost class.

__/__/____

1) WAP to demonstrate how to declare, define and call a constructor member func^{n} in a nested class & show the ~~firing~~ firing order of nested class.

abc will get very first memory followed by x, y and z.

```
#include <iostream>
using namespace std;
class abc
{
    Public : abc ();
        class x
        {
    Public : x ();
        class y
        {
        Public : Y ();
            class z
        {  Public : z ();
        };
    };
};
abc :: abc ()
{
cout <<" abc constructor";
}
abc :: x :: x ()
{
cout <<" x constructor ";
}
abc :: x :: Y :: Y ()
{ cout <<" Y constructor"; }
abc :: x :: y :: z :: z ()
{ cout <<" z constructor "; }
```

abc
↓
x
↓
y
↓
z

__/__/____

```
int main ()
  {
    abc obj ;
    abc :: x obj1 ;
  abc :: x :: y obj2;
abc :: x::y : : z obj3;
    return 0 ;
  }
```

DESTRUCTOR

1) It is a function that automatically execute when an object is destroyed.
2) It is executed whenever an instance of class to which it belongs goes out of existance.

Rules for syntax ↓

1) Destructor func name is same as that of class name except that 1st character of name must be a tilt operator. (~).
2) It is declared with no return types.
3) It can't be declared static, const or volatile.
4) It takes no arguments and therefore can't be overloaded.
5) It should have public access in class declaration.

can't have different arguments

Syntax.

```
class abc
{
Private : int a, b ;
Public : abc () ; //constructor
~abc () ; //destructor
};
```

```
int main ()
  {
    abc obj ;
    abc :: x obj1 ;
abc :: x :: y obj2;
abc :: x :: y :: z obj3;
    return 0;
  }
```

DESTRUCTOR

1) It is a function that automatically execute when an object is destroyed.
2) It is executed whenever an instance of class to which it ~~be~~ belongs goes out of existance.

Rules for syntax ↓

1) Destructor func name is same as that of class name except that 1st character of name must be a tilt operator. (~).
2) It is declared with no return types.
3) It can't be declared static, const or volatile.
4) It takes no arguments and therefore can't be overloaded.
5) It should have public access in class declaration.

can't have different arguments

Syntax:

```
class abc
{
private : int a, b;
public : abc ();  // constructor
         ~abc ();  // destructor
};
```

```
abc::abc()
  {
   cout<<"abc constructor";
  }
abc::~abc::~abc()
  {
   cout<<"abc destructor",
  }
}
```

#Code #Distructor in nested class.

A program to demonstrate, define & call ~destructor in nested class.

default constructor is already present in the program.

```
class abc
{
public:~abc()
    class x
    {
   Public:~x()
    class y
   {
   Public:~y()
    class z
  {
   Public:~z()
   };
  };
 };
};
~abc::~abc()
{ cout<<"~abc is destructor"; }
~abc::~x::~x()
  { cout<<"~x destructor"; }
```

__/__/____

```
~abc::~x::~y::~y()
{
cout<<"~y is destructor";
}
~abc::~x::~y::~z::~z()
{
cout<<"~z destructor";
}
int main()
{
abc obj;
abc::x obj;
abc::x::y obj;
abc::x::y::z obj;
}
```

destructor fire ⇒ destruction starts from inner most towards outermost

HW.

⇒ WAP to represent constructor and destructor in a single program. also show firing order of both.

obj()
~abc()

inline func^{n} → very important in C++.

friend func^{n}.

normal func^{n} with friend keyword.

Friend is a special mechanism for letting non-member func^{n}'s access private data. A friend func^{n} may be either declare or define within the scope of class definition. The keyword friend inform the compiler that it is not a member func^{n} of the class.

If the friend func^{n} is declared within the class it must be defined outside the class but shouldn't be repeated the keyword friend.

Syntax ↓

```
friend return type func^n name( );
```

program ↓

Program segment shows how a friend func^{n} is defined with in the scope of class definition.

```
#
using
class alpha
{ private : int a;
Public : void getdata ( )
        {
          cin >> a ; } class name parameter
friend void display (alpha abc)
        {
          cout << abc.a ;
        }.
};
        int main ( )
{ alpa obj;
        obj . getdata ( );
calling friend func^n → display ( obj );
        }
```

Program segment shows how a friend func^{n} is displayed out of the class definition.

```
class alpha
{
  private : int a;
  public : void getdata()
           {
             cin >> a;
           }
  friend void display();
};
void display(alpha abc);// using keyword friend multiple times gives error.
{
  cout << abc;
}
```

A friend func^{n} can be declare through public or private access specifier.

Find the largest of 2 numbers using friend func^{n}.

```
#
using
class numbers
{
  Public: int a, int b;
          void getdata();
  {
    cin >> a >> b;
  }
  friend void display(number abc);
};
```

```
void display (numbers a&b)
{
    if (abc.a > abc.b)
    { cout << "a is greater";
    }
    else
    {
    cout << " b is greater ";
    }
}

int main()
{
    numbers obj1;
    obj1.getdata();
    display (obj1);
}
```

Mean of 5 numbers. [friend func^n]

```
class mean
{
    int a, b, c, d, e, M;
public: void getdata()
    {
    cin >> a >> b >> c >> d >> e >> m;
    }
friend void display (mean xyz);
};
void display (mean xyz)
{
obj.m = (obj.a + obj.b + obj.c + obj.d + obj.e) / 5
cout << "mean of 5 no.s is :" << obj.m << endl;
}
int main ()
```

{

mean obj1;
obj1 get data ();
display(obj1);
}

__/__/____

OR

{we can use for loop in case of many numbers}

Example of forward declaration as well as

__/__/____

Write a program to show how a friend func^n works as a bridge b/w the classes.

forward declaration

```
class ABC //declared forward to avoid error
class xyz
{
private : int x;
public : void setdata (int a);
          {
            x = a;
          }
friend void max (xyz, ABC); //friend func^n for xyz
};
class ABC
{
  int y;
  public : void setdata (int b);
           {
             y = b;
           }
friend void max (xyz, ABC); //friend func^n for ABC
};
void max (xyz t1, ABC t2)
{
  if (t1.x > t2.y)
    cout << t1.x;
  else
    cout << t2.y;
}
int main()
```

```
{
    ABC abc;
    XYZ xyz;
  abc setdata (10);
  xyz setdata (20);
  max (xyz, abc);
  }
```

The func^n max has arguments from both XYZ and ABC when the func^n max is declared as a friend in XYZ for the first time the compiler will not acknowledge the presence of ABC unless its name is declared in the begining as class ABC. This is known as forward declaration.

HW (Swapping)

How to use a common friend func^n to exchange the private value of 2 classes.

Changes in private func^n leads to oppose the concept of data hiding.

Friend Classes :→

It can access private and protected member of the other class in which it is declared as a friend.

It is sometimes useful to allow a particular class to access private members of another class.

Syntax

```
class node
{
int key;
node * ptr;
friend class link list;
};
```

Program →

```
class xyz || no need of forward declaration
{
int a;
public : void set value ();
{
x = 10;
}
friend class ABC;
}
class ABC
public void display (xyz& t)
{
cout << t.x;
}
int main ()
```

class type value

whether it is space

classname

object name

__/__/____

```
{
    XYZ xyz;
    ABC abc;
    abc.display(a);
}
```

unary, binary, ternary

#OPERATOR OVERLOADING

we use primitive data type for operators.

user defined data type for operators is called operator overloading

1) C++ allows you to specify more than one (different data type) definition for an operator in the same slope is called operator overloading.

2) We can overload many built in operators available in C++

3) It is a type of compiled time polymorphism in which an operator is overloaded to give user defined meaning to it.

4) Some operators can't be overloaded.
Example, scope, sizeof, members elector i.e dot operator, member pointer elector (*), and ternary operator (?:)

__/__/____

RULES →

1) Only existing operators can be overloaded.

2) The overloaded operator must have atleast one operand i.e of user defined type. (for binary)

3) We can't change the basic meaning of a operator

syntax :

maybe a class ↓ keyword ↓

return type operator sign (parameters)
{
}

4) Operator which are overloaded can't be ~~op~~ over readen.

5) There are some operators that can't be overloaded Find out the list and why?

6) Binary operators are overloaded through a member func^{n}. Take one explicit argument and those which are overloaded through a friend func^{n} take 2 explicit arguments.

(1) Binary operator overloading [use of 2 operands]

```
class complex
{
int real, imag ;
complex (int r, int g) // constructor
 {
  real = r ;
  imag = i ;
 }
complex operator + (complex ob)
{
  complex res ;
  res. real = real + ob. real ;
  res. imag = imag + ob. imag ;
  return res ;
 }
void print ( )
};
{
 cout << real << "+" << "i" << imag;
```

object name
primitive
object
returning class type

(2) overload increment operator } unary

```
class increment
{
int value;
increment : value (10) { }
{ void operator ++ ( )
 {
  ++ value ;
 }
cout << value ;
```

__/__/____

```
};
main()
{
increment i;
  ++i;
}
```

3) write a program to overload unary minus operator

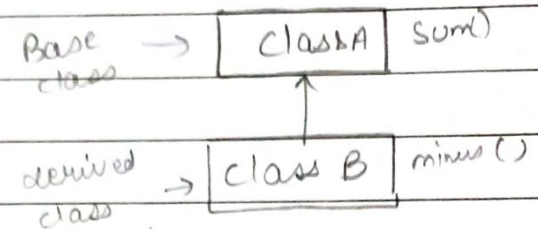
```
class Unary
{
int x,y,z;
void getdata(a,b,c)
{
 x=a;
 y=b;
 z=c;
```

```
class Unary
{
 int value;
 Unary::value(20){}
 void operator -()
 {
    -value;
 }
 cout << value;
};
```

__/__/____

INHERITANCE →

the mechanism of driving a new class from arold class is known as inheritance.



Reuseability of code in inheritance } advantage.

Syntax }

```
class derived classname : Access mode Base classname
{
}
```

Ex => Class B : public A ;

single inheritance

```
class A
{
  public : int a = 10;
           void display_a ()
           {
             cout << a ;
           }
};

class B : public A
{
  Public : int b = 20;
           void display_b ()
```

```
{
  cout << b;
}}
};
  main()
  {
   Bobj();
  obj.display_b();
  obj.display_a();
}
```

Write a program for 2 classes of sum and minus and call through one class.

```
class sum
{
  public: int a, b;
public: void display sum(int x, int y).
    {
      a = x;
      b = y;
    }
    {
      cout << x + y;
    }
};
class minus : public sum
{
  int c, d;
public: void display minus (u, v)
```

```
{
    c = u;
    d = v;
    cout << u - v;
  }
};
int main()
{
  minus obj();
  obj.display sum (6, 10);
  obj. display minus (12, 3);
}
```

set ptr = head

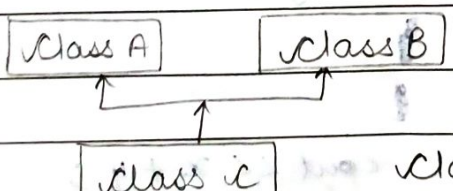
head = head [next]

free ptr

Class B : public A

↓ access mode

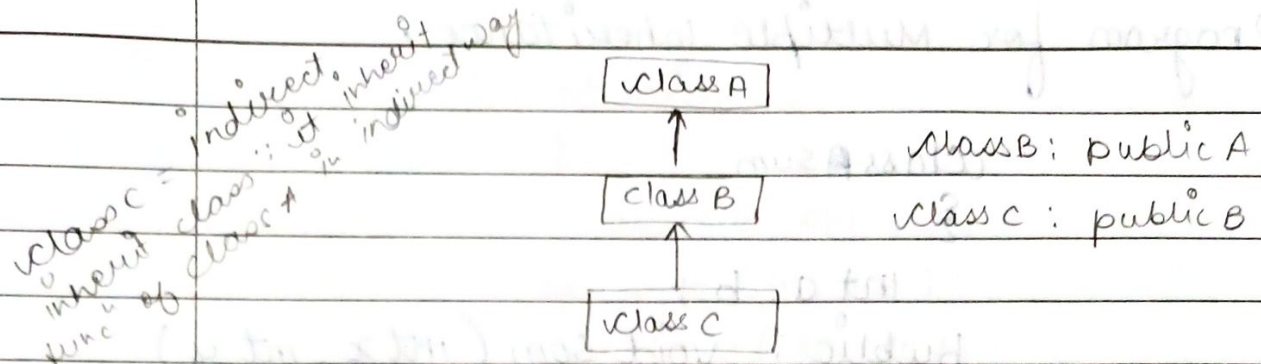
a) Single Inheritance



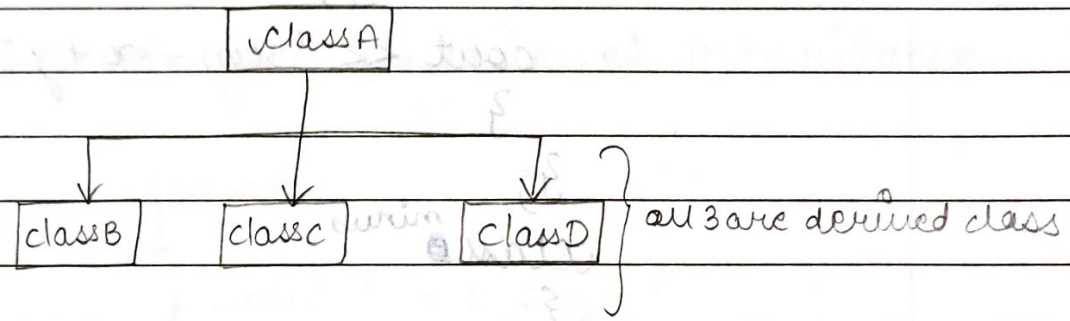
Class C : public A, public B

b) Multiple Inheritance

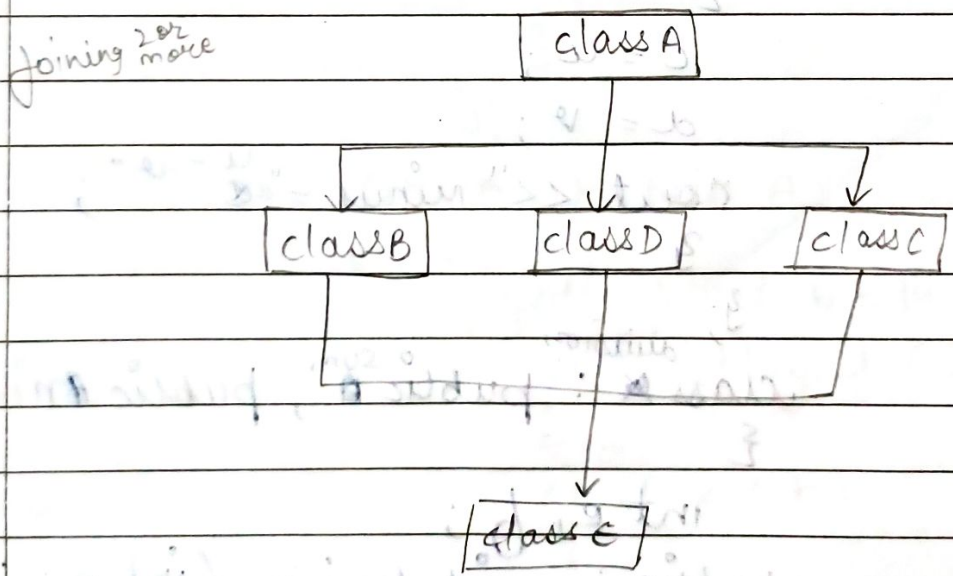
__/__/____



c) Multilevel Inheritance



d) Hierarchal



e) Hybrid.

__/__/____

① Program for Multiple Inheritance.

```
class sum
{
   int a, b;
public: void sum (int x, int y)
   {
      a = x;
      b = y;
   cout << "sum =" << x+y";
   }
};
class minus
{
   int c, d;
public: void minus (int u, int v)
   {
      c = u;
      d = v;
   cout << "minus =" << u-v";
   }
};
class division : public sum, public minus
{
   int e, f;
public: void division (int p, int q)
   {
      e = p;
      f = q;
   cout << "division =" << p/q";
   }
};
```

```
int main ()
{
   division obj ();
obj. sum (10, 12);
obj. minus (20, 10);
obj. division (40, 4);
}
```

(2) Program for Multilevel Inheritance.

```
class A
{
public : int a = 4;
    void display_a()
    {
    cout << a;
    }
};
class B : public A
{
    public : int b = 14;
    void display_b ()
    {
    cout << b;
    }
};
class c : public B
{
    public : int c = 44;
    void display_c ()
```

```
{
 cout << c;
}
};
int main()
{ C obj;
 obj.display_a();
 obj.display_b();
 obj.display_c();
}
```

(1) Write a program in which B is a base class D is derived class. The class B contains one private data member, one public data member and 3 public member func^{n}. and class D contain 1 private data member and 2 public member func^{n}.

Soln →

```
#include <iostream>
using namespace std;
class B
{
 int a;
public : void display(int b)
 {
  a = b;
  cout << b;
 }
}; void sum()
{ cout << b + 10; }
void diff()
```

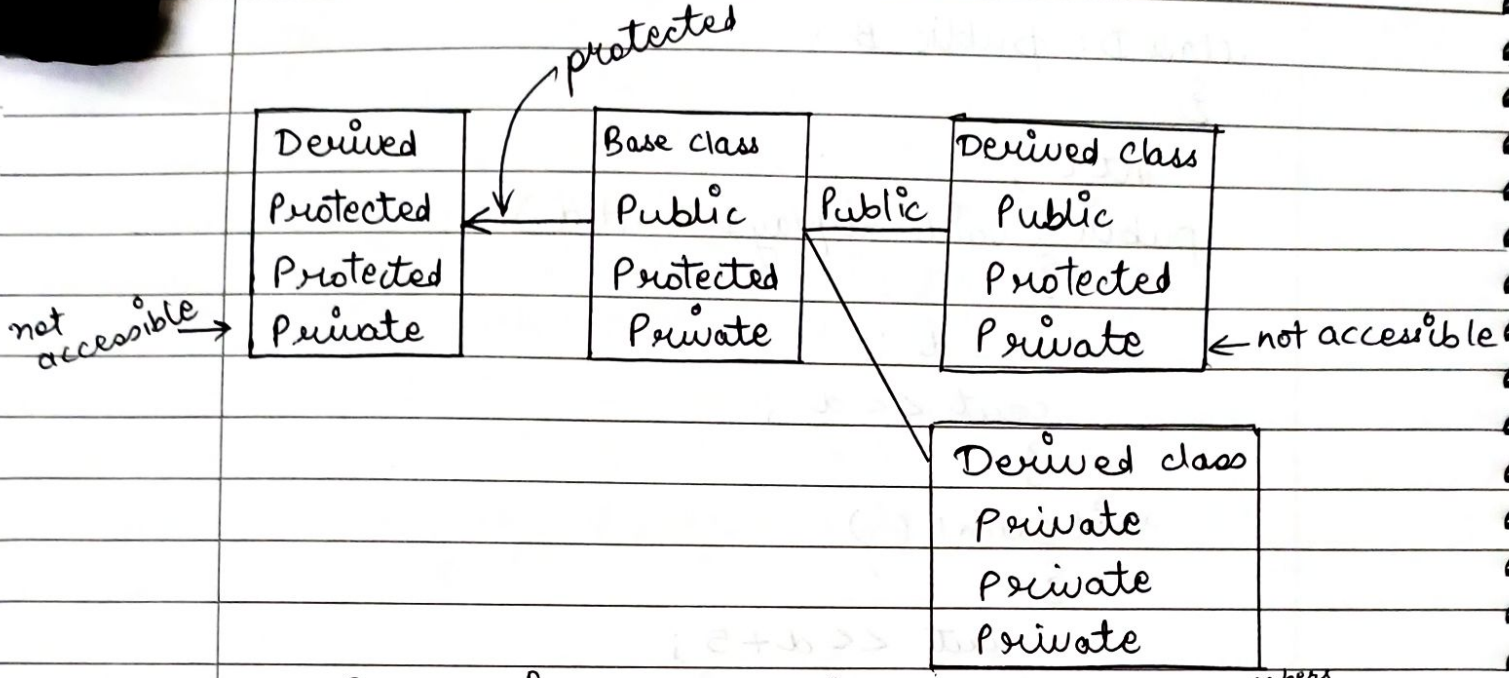
```
{
cout << b-10;
}
```

__/__/__

```
class D: public B
 {
   int c;
 public: void display1 (int d)
         {
          c = d;
          cout << d;
         }
  void sum1 ()
         {
           cout << d+5;
         }
  [illegible]
  {
    [illegible]
  }
};
  int main ()
```

cant access private members in inheritance

Access Mode in Inheritance



types of inheritance on the bases of access mode

Public Inheritance → it makes public members in base member in public member of derived class and public member of base member protected in derived class.

Private Inheritance → it makes public and protected members of base class to private member in derived class, whereas private members remain as it is in derived class.

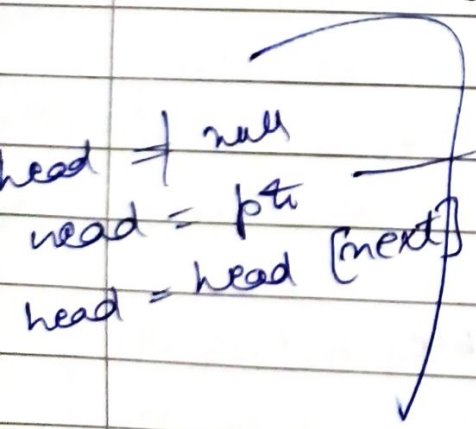
Protected Inheritance → it makes public member of base class to protected members in derived class and procted and private remains as it is in derived class but private members becomes not accessible.

__/__/____

```
class A
{
    public : int a;
    private : int b;
    protected : int c;
};
  class B : public A
  {
      // a = public
      // b = not accessible
      // c = protected
  };
  class C : private A
  {
      // a = private
      // b = private
      // C = private
  };
  class D : protected A
  {
      // a = protected
      // b = not accessible
      // c = protected
  };
```

Protected inheritance
Private inheritance

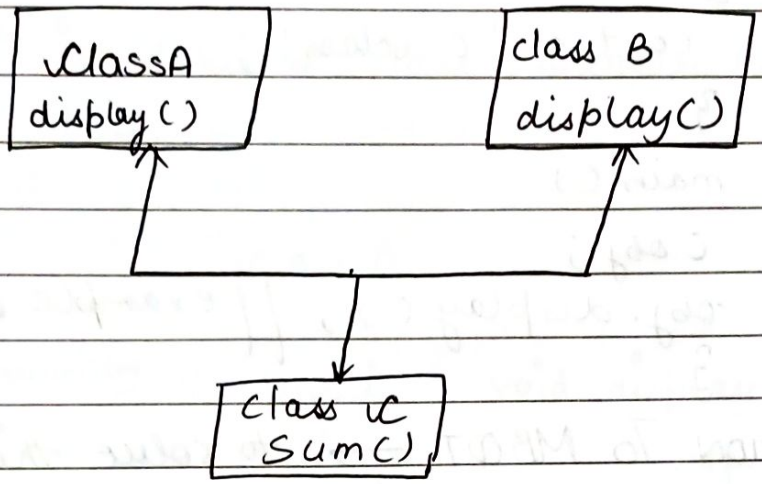
```
class A
{
  private : int a = 5;
  protected : int b = 10;
  public : int c = 20;
          int display_a ( ) // accessing private
          {
            return a;
          }
};
class B : public A
  {
    public: int display_b ( ) // accessing protected
          {
            return b;
          }
  };
          main ( )
          {
          B obj;
          cout << obj.c;
          cout << obj.display_a ( );
          cout << obj.display_b ( );
          }
```



MBQT

__/__/____

Same name different class.



C obj;

obj.display(); // Problem is same name then call to which class.?

```
classA
{
 Public:
        void display()
        {
          cout <<"A calling";
        }
};
 classB
 {
    Public: void display()
       {
         cout <<"B calling";
       }
 };
classC : publicA, publicB
 {
    Public: void sum()
```

appears only in Multiple inheritance

__/__/____

```
{
  cout << " C class ";
}
main()
{ obj;
obj.display(); // example of MBOT
}
```

SOLUTION TO MBOT → to solve this scope resolution operator is used.

Syntax:

objectname.classname :: func^n name.

Removing MBOT

```
obj.A::display();
obj.B::display();
}
```

MBOT :→ Same name of func^n of different base class. } MBOT

__/__/____

Function Overwritting.

Func^{n} in derived class overwrites the func^{n} in base class.

Same name of base class and derived class. } overwriting

```
class A
{
public : void display ()
{
cout << " Base class ";
};
class B : public A
{
public : void display ()
{
cout << " Derived class ";
}
};
void main ()
{
B obj. ;
obj. display ();
```

output => Derived class

Here the same func^{n} display is defined in both derived & base class, so when we call display func^{n} by object obj of derived class. Display from derived is executed by overwriting in base class

__/__/____

Access overwritten func^{n} to the base class.

```
class A
{
public : void display()
  {
  cout << "Base class";
  }
};
class B : public A
  {
  cout << "Derived class";
  }
};
void main()
{
  B obj;
obj.A :: display(); // resolving problem of overwriting
  }
```

→ To access the overwritten func^{n} of base class we use scope resolution operator.

→ We can access the overwritten func^{n} by using a pointer of the base class to point to an object of the derived class and then calling the func^{n} from that pointer.

[type of Polymorphism]

__/__/____

Func^{n} Overloading

it is a logical method of calling several func^{n} with diff. arguments & data type that perform basically identical things by the same name.

assignment,
header file
↓
func^{n} name
↓
use of func^{n}

void sum (int a, int b)
int sum (int a, float b) — change in data type
void sum (float a, float b, float c)

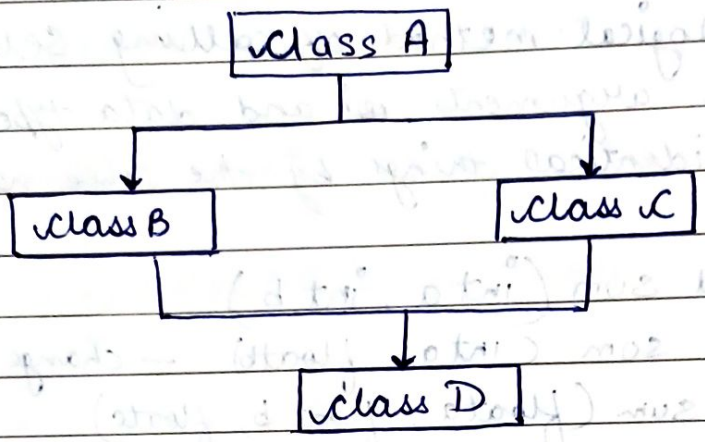
~~class A~~
~~{~~
~~public: void sum (int a, int b)~~
~~int sum (int a, float b)~~
~~void sum (float a, float b, float c)~~

[# constructor / Destructor in inheritance ?]

Advantages =① eliminating the use of diff. func^{n} name for the same operation.
② helps to understand debug [erase removing] easily.
③ Easy maintainability of code
④ Better understanding of the relation b/w the program & outside world.

__/__/____

Virtual Base Class



D → B → A } compiler get confused to follow
D → C → A } which route. hence we get AMBIGUITY error.

Virtual Base Classes are used to prevent multiple instances of a given class from appearing in an inheritance hyrarichy when using multiple inheritance.

Base classes are the classes from which other classes are derived. The derived classes have access to variables and method of base class. This structure is known as inheritance hirarichy.

Virtual Class : it is defined by writing a keyword 'virtual' in the derived class which allow only one copy to data to be copied to the derived class. It prevents multiple instances of a class appearing as a

parent class in the inheritance hierarchy when multiple inheritance is used.

Need for virtual classes.

```
#
using namespace std;
class A
{
 private : int a
 Public : void getdata (A=5)
          {
            a = A;
          }
};
class B : Public A //Public virtual A
 { };
class C : public A //Public virtual A
 { };
class D : public B, public C
 { };
Void main ()
{
 D. obj;
 obj. getdata ( ); // error, need of virtual classes. shows AMBOT
}
```

How to declare virtual base class in C++?

__/__/____

Syntax →

class name of base class : virtual access specifier name of derived class.

class B : virtual public A

OR

class B : public virtual A

→ To prevent the error and let the compiler work efficiently we have to use virtual base class when multiple inheritance occur.

→ It saves space and avoids MBOT.

→ It prevents duplication of its datamember.

→ Only one copy of its data member is shared by all the derived classes that use virtual base class.

Pure Virtual Func^{n} → to keep empty func^{n} in base class.

~~declare~~ ~~class~~

It is a func^{n} that does nothing which means that you can declare pure virtual func^{n}

__/__/____

in a base class that doesn't have a description in a base class.

Syntax →

virtual returntype func name()
{ }

OR

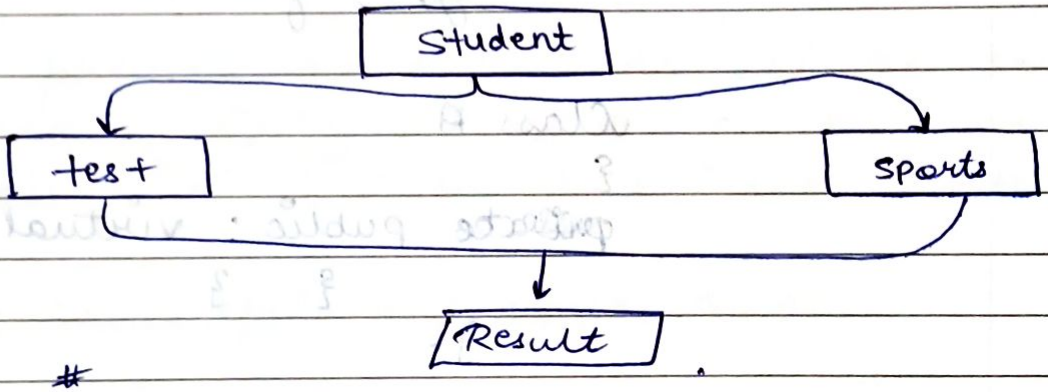
virtual return type func name() = 0;

```
Class A
{
public: virtual void colour()
        { }
};
Class B : public A
{
public: void colour()
{
  cout << "Red" << endl;
}
};
class C : public A
{
public: void colour()
{
  cout << "Green" << endl;
}
};
void main()
{
  B obj; obj.colour;
```

__/__/____

```
c obj1;
obj. colour ;
}
```

Write a program →



using . . .

```
class student
{
public : char name[10] = "Akriti" ;
         int rollno = 12100723 ;
void display()
  {
    cout << " Name of the student : " << name << endl;
    cout << " Roll no. of the student : " << rollno. << endl;
  }
};
class test : public virtual student
  {
  public :
    int Maths = 96 ;
    int c++ = 91 ;
        void display()
```

```
{
cout <<" Marks of Maths: " << Maths << endl;
cout <<" Marks of C++ : " << C++ << endl;
}
};
class Sports : public virtual student
{
  public : void marks ()
    int cricket = 100 ;
    int Baseball = 70;
  & void display ()
  {
  cout <<" Scores of sports in cricket : "<<cricket<<
                                          endl;
cout <<"Scores of sports in baseball : " << baseball <<endl;
  }
};
void main () class Result ; public test, public sports
  { {
public : void getdata () display data ()
  {
cout <<" Result of student:" <<'Name : " << name <<
"Roll no.:" << rollno. <<"Marks in maths :" << maths <<
"Marks in C++ :" << C++ << "score in cricket:" << cricke
<< "score in baseball :" << baseball << endl;
  }
};
void main ()
{
  Result obj;
obj. student obj.student :: display ();
obj.test :: display (); obj.sports :: display (); }
```

area of circle

πr^2

__/__/____

```
class area
{
 int radius;
 public: void get_data (int r)
    {
      radius = r;
    cout <<" radius :" << radius;
    }
     void display ()
    { int π = 3.14;
int a = π radius * radius;
    cout << " a: " << a;
    }
    };
     int main()
     {
   area obj;
    obj. get data (12);
    obj. display();
return 0;}
```

__/__/____

Polymorphism :

compile time → func^n overloading, operator overloading.

Run-time

one name, multiple forms

The concept of polymorphism is implemented using concept of func^n overloading and operator overloading.

The overloaded member func^n are selected for invoking by matching arguments, type and number.

```
sum (int a, int b);
sum (float a, float b);
```

This information is known to compiler at compile time. This is called Early Binding Static Binding or Static Linking.
Also it is known as Compile time polymorphism

* Compile Time → object is bound to its func^n call at compile time.

Run Time → If the appropriate member func^n could be selected while the program is running is called as Run Time polymorphism.

Run_Time is achieved through virtual func^n's.

* Late binding * Dynamic Binding.

(*ptr.)

__/__/____

Dynamic Binding requires the pointers to object.

Pointer ⇒ int *ptr, a;
ptr = &a;

Pointers with array and strings

Pointers to func^n

Pointers to object

This Pointer "→"

```
class Box
{
private : int l, b, h ; //instance variable
public : void setdata (int x, int y, int z)  // local variable
  {
    l = x, b = y, h = z ;
  }
  void display ( )
  {
    cout << l << b << h ;
  }
};
int main ( )
{
Box obj1;
obj1.setdata (10,20,30);
obj1.display ( );
}
```

Properties of this pointer with advantages.

Pointers to derived class

```
class B
{ };
class D : public B
{ };
int main()
{
  B obj_b;
  D obj_d;
  B *ptr;
  ptr = &obj_b;
  ptr = &obj_d;
  return 0;
}
```

Virtual func^{n} ⇒ • a virtual func^{n} is a member func^{n} which is declared with in base class and re-define by derived class.

• When you reffer a derived class to an object using a pointer or a reference to the base class, you can call virtual func^{n} for that object and execute the derived class version of the func^{n}.

• Virtual func^{n} ensures that correct func^{n} is called from an object regardless of the type of reference (pointer) used for func^{n} call.

__/__/____

- They are used to achieve run time polymorphism
- Func^n are declared with a virtual keyword in base class.
- The resolving of func^n call is done at runtime

```
class B
{
Public : void show ( )
  {
cout << "B class shows func^n ";
  }
virtual void print ( )
  {
  cout <<"B class prints func^n" ;
  }
class D : public B
  {
  public : void show ( )
    {
  cout << "D class show" ;
    }
  void print ( )
  {
  cout <<"D class print func^n";
  }
  };
    int main ( )
    {
```

__/__/____

through pointers.

```
B obj-B;
B *ptr;
ptr = & obj-B
D obj-d;
ptr = & obj-d
```

ptr → show()
ptr → print()

Properties of virtual func^n:	Rules for virtual func^n

* How polymorphism is achieved whileRuntime ⇒ virtual func^n. concept of inheritance.

Abstract classes:

A class is Abstract if it has atleast 1 pure virtual func^n.

We can have pointers and reference of abstract class type.

Abstract class can't be initiated or object of abstract class. If we donot overwrite pure virtual func^n in derived class then derived class becomes abstract class

we can't make

Ex⇒

```
class A
{
public: virtual void display ();
{
[illegible];
}
};
class B: public A
{
public: void display ();
```

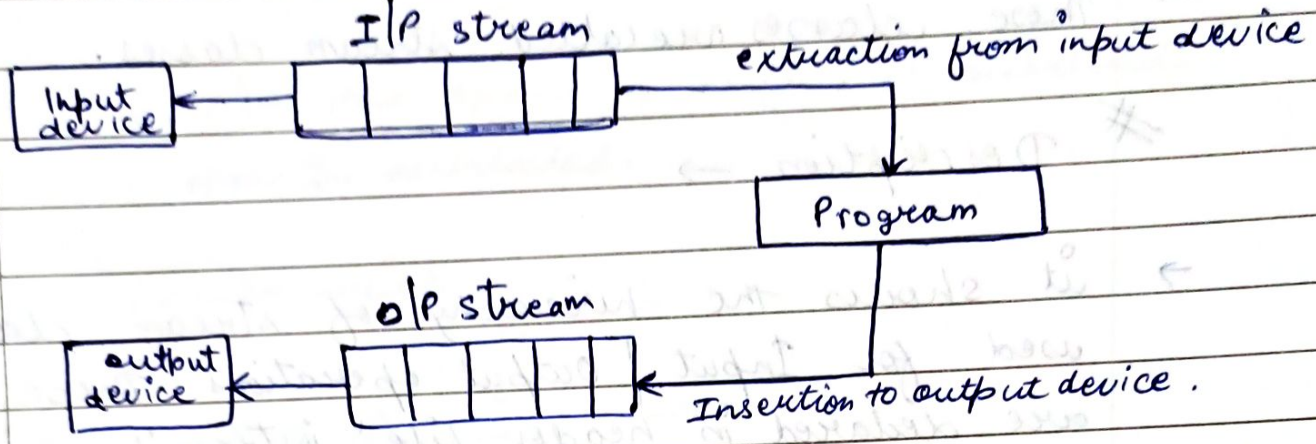
```
{
  cout << " Red " << endl;
}
};
class C : Public A
 {
 public : void display ()
 {
 cout << " Green " << endl;
 }
};
void main ()
{
  B obj;
  obj.display;
   C obj1;
  obj1. display;
 }
```

__/__/____

C++ Streams:

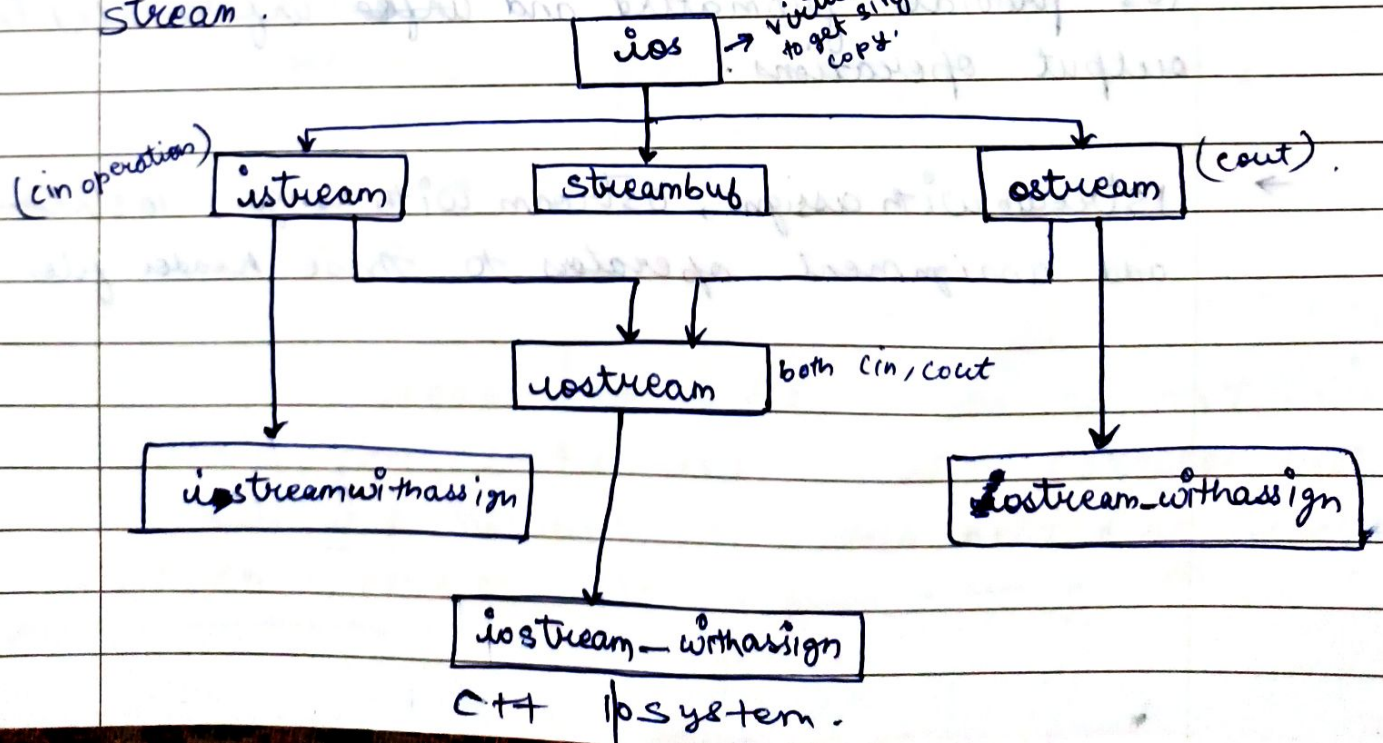
Managing Input-output operations ⇒

Streams - Sequence of bytes



→ It acts as a source from which the input data can be obtained as a destination to which output data can be set.

→ The source stream that provides data to the program is called input stream and destination stream that recieves output from the program is called output stream.



→ It contains a hierarchy of stream that are used to define various streams to deal with various disk files.

→ Those classes are called stream classes.

Description →

→ It shows the hierarchy of stream classes used for Input / output operations. These classes are declared in header file "iostream".

→ Ios is the base class for istream and ostream which is a base class for inter base class for iostream. The class ios is declared as virtual base class. So that only 1 copy of its members are inherited by the iostream.

→ Ios provides fromatted and unfo unformated input output operations.

→ Istream with assign, ostream with assign, iostream with assign add assignment operators to these header files

__/__/____

Unformated various I/o operations →

explain → (cin) (cout).] objects of class iostream.

cin >>

cout <<

(>>) → this operator reads, it overloads.
operator overloaded.

(<<) → write

Get and Put functions →

```
Void main ()
{
Char c;
cin.get(c);
While (c!='\n')
{
cout.put(c);
cin.get(c);
}
```

output:- HELLO
HELLO

```
char c;
cin.get(c);
cout.put(c);
```

output : HELLO
H

This code reads and display a line of text. Cin operator can also be used to read a character but it will skip the wide space and new line character

| /n = new line | next line |

__/__/____

to count how many character we have →

```
Void main ()
{
 Char c;
 Int count = 1;
 cin.get(ic);
 while (ic != "\n")
 {
   cout.put(ic);
   cin.get(ic);
   count++;
 }
   cout << count;
```

output :→
input given { HELLO
HELLO5

output →
input { HELLO WORLD
HELLO WORLD 11

Getline and write funch.

```
cin.getline (string's name, sizeof);
cout.write (name, size);
```

#

```
void main ()
{
 char city [20];
 cin >> city;
 cout << city;
}
```

__/__/____

Formatted Console I/O operation

- ios class function & flags
- Manuplators
- user defined o/p func^n (ostream & manuplators) (ostream & output)

→ Width (w) → no. of columns.
cout.width(w)
setw(int a)

→ precision (p)
cout.precision(p)
setprecision(int a)

→ fill()
cout.fill('*')
setfill()

→ setf()
cout.setf(arg1, arg2)
cout.setf(ios::left, ios::adjustfield)
setiosflags(long f)
setf(ios::justification)
justification

unsetf

write func^n →

```
char city[20];
int size = 20;
cout << "Enter:";
① cin.getline(city, size);
② cout.write(city, 3);
```

output :
Enter :
by user → New Delhi

output of ① = New Delhi

output of ② = New

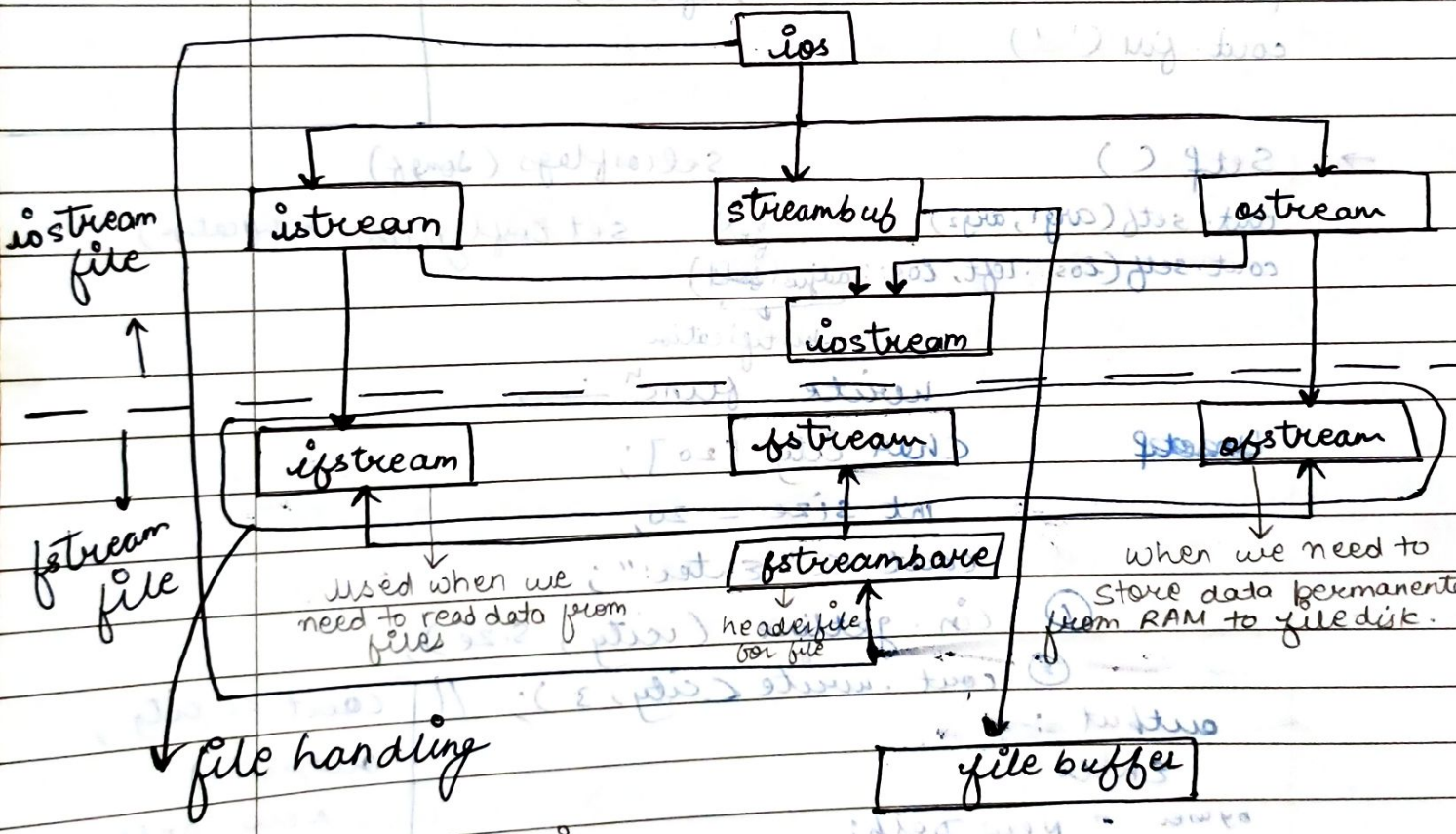
```
cout << city;
```
output →
New Delhi
New Delhi

Program for → __/__/____

result

Product **	Price **	Total Value
10	50	
15	10	
20	5	
25	5	
30	5	
	Total	

Working with files



ifstream fin

fin >>

; just like cin but we can make any object

Stream classes for file operation

```
#include <iostream>
#include <fstream>
using namespace std;
main()
{
```

to store in file →

```
ofstream fout;
fout.open("first file.txt");
fout<<"Why would I, Sukhjeevan's first file";
fout.close();
fout.open("2nd file.txt");
fout<<"I am a potato, Sukhjeevan's 2nd file";
fout.close();
int size[80];
char line[size];
ifstream fin;
fin.open("first file.txt");
while(fin)
{
fin.getline(line, size);
cout<<line;
}
fin.close();
```

when to get the file back in program →

```
fin.open("2nd file.txt");
while(fin)
{
fin.getline(line, size);
cout<<line;
}
fin.close();
}
```

getline helps in reading whole line with spacing.

__/__/____

Detecting End of the file:

By using statement ((while (fin))) an ifstream object such as 'fin' returns the value of zero if any error occur in the file operation including the end of file. while loop terminate when 'fin' returns a value of zero on reaching the end of file condition.

to check end →.

```
if (fin.eof()!=0)
{
    exit()
}
else
ptr = address of new node
ptr [next] = head.
ptr [info] = ITEM
head = ptr
exit.
```

End of file {eof()} is a member func^n of ios class.
It returns a non-zero values {1 | true} if the end of file condition is encountered.

__/__/____

File Modes:

(1) ios::app
append to end of file.

(2) ios::ate
goto end of file on opening.

(3) ios::binary
file opens in binary mode

(4) ios::in
open file to read.

} by default

(5) ios::nocreate
open fails if file doesn't exist.

(6) ios::noreplace
open fails if file already exists.

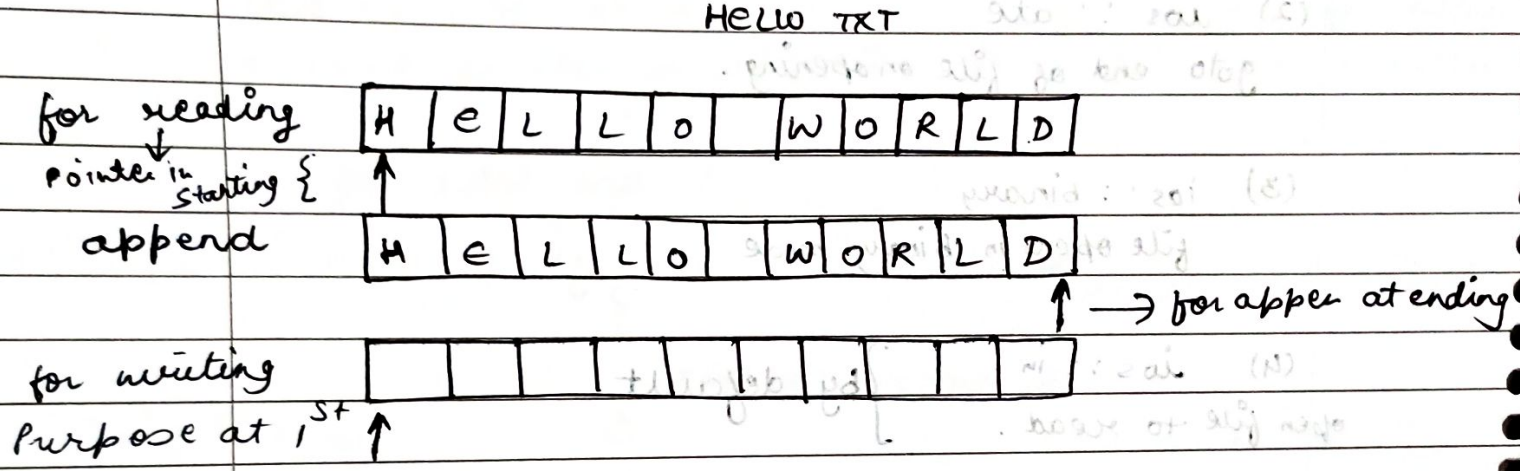
(7) ios::out
open file to write

} by default.

(8) ios::trunc
delete the contents of file if it exist.

File Pointers

→ Input Pointer
→ Output Pointer



File has associated pointers known as file pointer
2 types → Input pointer / Get pointer.
→ it is used for reading the contents of a file.

→ output pointer / put pointer
it is used for writing content

Default actions ⇒
→ When we open a file in read only mode the input pointer is automatically set at the starting.

→ when we open the file in writing mode only pointer is set at start. And

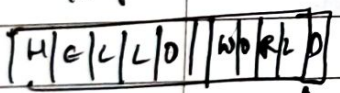
__/__/____

→ And if we want to open existing file in append mode to add more data then pointer is set to end of the file.

Functions for manuplation of file pointer.

(1) seekg()	(2) seekp()	(3) tellg()	(4) tellp()
Moves get pointer to a specified location	Moves put pointer to specified position	to get actual position in reading of get pointer	to get actual position in writing of put pointer.
infile.seekg(10); (position where we need pointer) — infile: any object; 10: ∴ pointer at 11th position	infile.seekp()		

in file




```
ofstream.fileout;
fileout open ("hello.txt", ios::app);
int p = fileout.tellp();  // it will print no.
output = 10
```

Specifying offset →

fout.seekg(0, ios::beg);	Go to start
fout.seekg(0, ios::cur);	
(0, ios::end);	
(m, ios::beg);	Move to (m+1)th byte in file
(m, ios::cur);	
(-m ios::cur);	Go backward by m byte from the current mode
(-m, ios::end);	

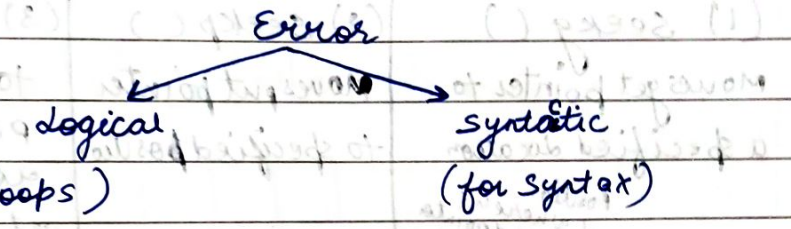
m = natural number to represent position / natural number

seekg (offset, reposition)

seekp (offset, reposition)

ios::beg.

EXCEPTION HANDLING



→ when user gives wrong input.

→ An exception is an unexpected problem.

→ Exception handling mechanism to transfer control from one ~~control~~ program to another

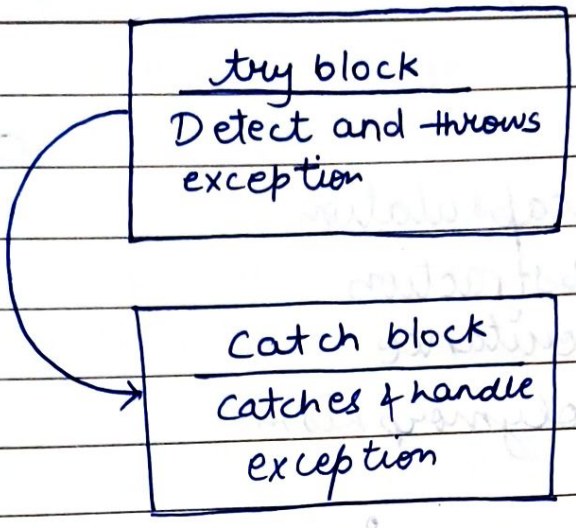
Blocks are used

TRY : A block of code which cause exception is placed inside the try block.

Catch: This block catch the exception thrown from the "try" block. Code to the exception is written inside this catch block.

: A program throws an exception when a problem shows up

__/__/____



#

```
int main()
{
int numinator, denominator, result;
cout <<" Enter numinator & denominator, result;
cin >> numinator >> denominator;
try {
      if (denominator == 0)
      throw [denominator] variable
      result = numinator / Denominator;
}
catch ( int exp )
      { cout <<" deno is 0 " <<exp;
      }
cout <<" Denominator is: " << result;
}
```

Exception, 3 blocks, Multiple catch mechanism.

OOPS →

① concept of classes and objects

class = Blueprint

4 principles →

1. Encapsulation
2. Abstraction
3. Inheritance
4. Polymorphism.

class :→ user defined

object is a instance of class.

Access specifier →

Public : can be accessed throughout the class.

Protected : can be accessed by derived class.

Private : can't be accessed by func's. or outside the class

Inheritance

Encapsulation

By default = private

↓

if no specifier is mentioned.