C++

For Interview Preparation

By: Deepa Chaurasia

https://www.linkedin.com/in/deepa-chaurasia-notes

Table of Contents

https://www.linkedin.com/in/deepa-chaurasia-notes

DIFFERENCE BETWEEN PROCEDURE ORIENTED PROGRAMMING AND OBJECT ORIENTED PROGRAMMING

	Procedure Oriented Programming	Object Oriented Programming
Divided into	In POP, program is divided into small parts called functions	In OOP, program is divided into parts called objects.
Importance	In POP, importance is not given to data but to functions as well as sequence of actions to be done.	In OOP, importance is given to the data rather than procedure or functions because it works as a real world.
Approach	POP follows Top down approach	OOP follows Bottom up approach.
Access Specifiers	POP does not have any access specifier	OOP has access specifiers named Public, Private, Protected, etc.
Data Moving	In POP, Data can move freely from function to function in the system.	In OOP, objects can move and communicate with each other through member functions.
Expansion	To add new data and function in POP is not so easy.	OOP provides an easy way to add new data and function.

Data Access	In POP, most function uses global data for sharing that can be accessed freely from function to function in the system.	In OOP, data can not move easily from function to function, it can be kept public or private so we can control the access of data.
Data Hiding	POP does not have any proper way for hiding data so it is less secure.	OOP provides Data Hiding so provides more security.
Overloading	In POP, overloading is not possible.	In OOP, overloading is possible in the form of function overloading and operator overloading.
Example	Example of POP are : C, VB, FORTRAN, Pascal.	Example of OOP are : C++, JAVA, VB.NET, C#.NET.

Concept OOP

(1) Class → collection of similar digits
↳ work as data type

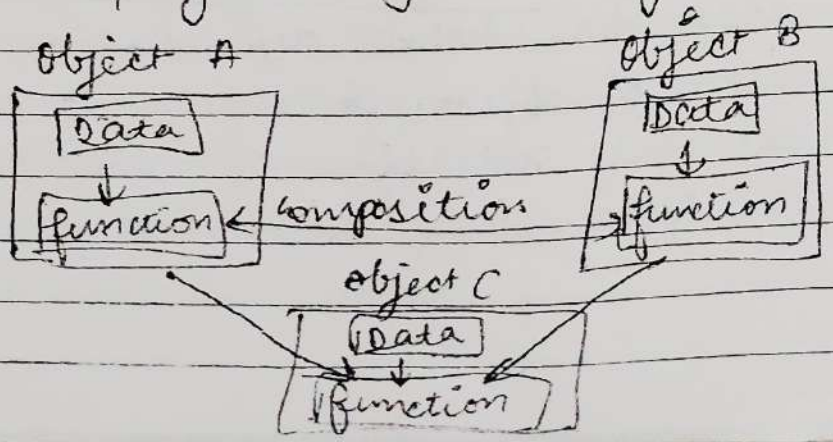
(2) Objects → real world entity { data, func^n
↳ work as variables

Eg:
```
class      car
{
// data     name
            colour
            model
            price
// fns      start()
            stop()
}
```

* Object Oriented Programming:-

(i) OOP decomposes a problem into a number of entities called objects & then builts data and function around these objects.

(ii) The organisation of data & function in object oriented programming is as follows:-



(iii) striking features of OOP are :-

(a) inphasis is on data rather than procedure
(b) programmes are divided into objects
(c) data structure are designed such that they characterise the objects.
(d) function that operate on the data are tied together in a data structure.
(e) data is hidden & cannot be accessed by external functions.
(f) objects may communicate with each other through functions.
(g) it follows the bottom up approach in program design.

V. Imp

* Basic Concepts of OOP :-

1) Object

⇒ Objects are basic runtime entities, they may represent a person, a place, a bank account, etc.

⇒ Each object contains data & code to manipulate the data.

2) Class

⇒ Objects are variables of the data type class.

⇒ Once a class has been defined, we can create any number of objects belonging to that class.

⇒ Classes are the collection of similar objects.

⇒ Eg: BMW, Audi, they belong to the class car.

3) Encapsulation

⇒ The wrapping up of data & function into a single unit (class) is known as encapsulation.

⇒ The data is not accessable to the outside world.

imp ⇒ The insulation of the data from direct access by the program is called data hiding / information hiding.

4) Abstraction

⇒ Abstraction refers to the act of representing the essential features without including the background details / explanations.

⇒ Since the classes use the concept of data abstraction, they are known as abstract data types.

v. imp 5) Inheritance

⇒ Deriving the properties of one class into another.

⇒ The existing class is known as parent class / Base class & the new class is known as child / derived class

★ C++ supports 5 types of inheritance

6) Polymorphism

⇒ It means ability to take more than one form.

⇒ We can overload a method / an operator.

⇒ The process of making an operator to exhibit different behaviours in different instances is known as operator overloading.

⇒ Using a single function name to perform different tasks is known as function overloading.

7) 1) Dynamic Binding overloading
⇒ It is associated with polymorphism & inheritance (to be covered in Unit-3).

8) Message Passing
⇒ The data from all classes passes through messages.

Diff. b/w POP / OOP

Subject of diff.	POP (C)	OOP (C++)
Problem decomposition	It decomposes the main problem in small parts called functions.	It decomposes the main problem in small parts called objects.
Use of data	Most func^n use global data.	Each object controls its own data
Connection Of Parts	It connects parts of the program by parameter passing approach.	It connects the parts of the program by message passing approach.
Passing of Data	Data is passed from one func^n to another.	Data never gets passed from one object to another

security of data	Data is allowed to move freely & no techniques are available to secure the data.	Different access modi-fiers are available to protect the data (public, private &
Designing Approach	Top down approach for designing the program.	It follows bottom up approach for designing the programmes
Languages	C, FORTRAN, COBOL	C++, JAVA, etc.

C++

1/8

- C++ is an object oriented programming language. It was developed by Bjarne Stroustrup at AT & T Bell Laboratories in early 1980's (81, 82, 83). He combined the features of object oriented programming to the C language, therefore C++ is an extension of C

1979

- Initially this languages was known as C with classes & ++ is this represents increment operator

- C++ is superset of C, the most imp. facilities that C++ acts onto C language are classes, inheritance, func^n overloading & operator overloading

* Specifying a Class in C++

- Class is the most imp. feature of C++. It is the way to bind the data & its associated functions together.

- class BSc3
  {
  data members -- // data] class members
  member functions
  };

- Access Specifiers
  ↙ ↘
  Public Private

- When defining a class we are creating a new abstract data type.
- A class specification has two parts:

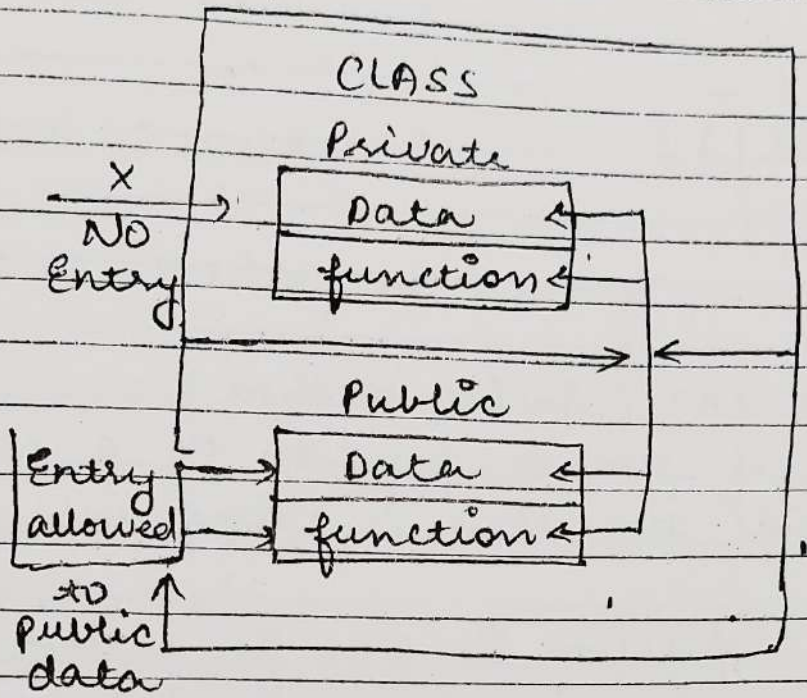
a) Class declaration
b) Class function definition

The General form of class declaration is :-

```
class class_name
{
    private;
        variable declarations;
        function declarations;
    public;
        variable declarations;
        function declarations;
};
```

- The key word class specifies the starting of the new abstract data type.
- The body of a class is enclosed in Braces, and is terminated by ;
- The class body contains declaration of variables and functions.
- These functions and variables are collectively called as class members.
- Class members are usually grouped under two sections :- (a) Private (b) Public
- The keywords private & public are known as access specifiers or visibility labels & are followed by a "colon :".
- The class members that have been declared as private can be accessed only from within the class.
- Public members can also be accessed from outside the class.
- By default, the members of class are private.
- If both labels are missing then by default all the members are private & such class is completely hidden from the outside world & does not serve any purpose.
- Variables declared inside the class are known as data members & the functions are known as member functions.
- Only the member functions can have access to the private data members & private member function
- The public members can be accessed from outside the class

* Data Hiding in Class :-



* Program :-

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

{
    class BSC
    {
      private;
       int id;
      char name[30];
      public;
     void getdata();
      void showdata();
};
void BSC :: getdata()
{
    cout << " Enter Roll No. ";
```

```
void main()
{
    A x;
    x.add();
    getch();
}
```

3/8
Imp

Q Inline function / Inline method ?

Ans: Definition of member function of a class when given inside the class is called inline function.

☆ method - a function in C++ is called method

☆ function - a function in C i.e. POP called function

* ~~Declaration~~ Defining Member function :-

(1) Outside the class

```
#include <iostream.h>
#include <conio.h>
class BSc
{
    Public;
        void study();
};
void BSc :: study()
{
    Cout << "Test on Monday";
}
Class BCA
{
```

```
    Public :
        void study();
};
void BCA :: study()
{
    cout << "Holiday on Monday";
}
void main()
{
    Bsc b1;
    BCA b2;
    b1.study();
    b2.study();
    getch()
}
```

(2) Inside the class

```
class Bsc
{
    Public :
    void study()
    {
        cout << "Test on monday";
    }
};
```

- Member functions can be defined in two places:-
a) Outside the class definition
b) Inside the class definition

a) Outside the Class Definition

→ Member functions that are declared inside a class have to be defined separately outside the class. The general form of member function definition is :-

* return_type class_name :: fn_name (list of arguments)
{
// body of method
}

→ The member skip label class name tells the compiler that the function "functionname" belongs to the class "class_name".
The symbol "::" is called scope resolution operator or class resolution operator.

b) Inside the Class Definition

→ Another method of defining a member function is to replace the function declaration by the actual function definition inside the class.

→ When a function is defined inside the class, it is treated as an "inline function".

(4) Fabonacci series

```
#include<stdio.h>
#include<conio.h>
void main()
{
    clrscr();
    int a, i, b, c, n;
    a=0; printf("Enter no. of elements in series");
    scanf("%d", &n);
    b=1;
    printf("%d; %d", a, b);
    c=a+b;
    a=b;
    b=c;
    for(i=1; i<=n; i++)
    {
        printf("%d", c);
    }
    getch();
}
```

6/8

Constructors

- Constructors are special member function whose task is to initialize the objects of its class.
- Whenever an object is created, the constructor of that class is automatically executed.
- A constructor is a special member function with the following properties :-

a) Name of the constructor is the same as class name
b) they do not have return types not even void so they cannot return data.

c) Constructors can be parameterized or non-parameterized.
d) They should be declared in public section.
e) They are invoked automatically at the time of object initialization (i.e. object creation).
f) Constructors cannot be inherited.

- A constructor is declared & defined as follows:-

```
class integer
{
    int m, n;
  public
      integer()
      {
      // constructor body
      }
};
```

* Program:-

```
class BSc2
{
    int roll;
    char name[30];
  public;
    BSc2( )
    {
      Cout << "Enter roll";
      Cin >> roll;
      Cout << "Enter name";
      Cin >> name;
    }
```

```
    void showdata()
    {
        cout << "Roll is" << roll;
        cout << "Name is" << name;
    }
};
int main()
{
    BSc 2  b1;
    BSc 2  b2, b3, b4;
    b1.showdata();
    b2.showdata();
    b3.showdata();
    return 0;
    getch();
}
```

- Role of Constructor :-

1.) Initialising values to the objects
2.) allocating memory to the object

Q- What is OOP ? Explain its concepts ?
Q- W.A.P to implement student class which stores & displays student information.
Q- Redo Q1 using concept of constructors
Q- W.A.P to print fabonacci series
Q- Write a program to input an integer value from keyboard & display on screen "WELL DONE" that many time.
Q- W.A.P to read values of a, b, c & display x where $x = a/b - c$

- User Defined Constructor :-

These constructors are defined by the programmer explicitly in a class. User defined constructor can be further characterized as :-

a) Non - parameterized
b) Parameterized
c) Copy constructor

⇒ Non - Parameterized Constructor :-

```
#include <iostream.h>
#include <conio.h>
class nisha
{
    int m, n;
    public:
     nisha()
     {
          m=0;
          n=0;
     }
    void show()
    {
        cout << m << n;
    }
};
```

```
void main()
{
    nisha x;
    x.show();
    getch();
}
```

Output:-

00

Parameterized Constructor :-

```
#include <iostream.h>
#include <conio.h>
class nisha
{
    int m, n;
    public:
    nisha(int x, int y)
    {
        m = x;
        n = y;
    }
    void show()
    {
        cout << m << n;
    }
};
void main()
{
    nisha a(3,5);
    a.show();
    getch();
}
```

Output :-

35

* Copy Constructor :-

```
#include <iostream.h>
#include <conio.h>
class copydemo
{
    int m, n;
    public;
     copydemo(int a, int b)
     {
          m = a;
          n = b;
     }
    copydemo (copydemo &x)
    {
          m = x. m
          n = x. m
    }
    void show()
    {
       cout << m << n;
    }
};
void main()
{
    copydemo c1(30, 40);
    copydemo c2(10,20);
    copydemo c3(c1);
      c1. show();
      c2. show();
    c3. show();
    getch();
}
```

Output :-

30 40

10 20

30 40

- A copy constructor is used to declare and initialize an object from another object. For example :- The statement copy const C3(C1) would define the object C3 and at the same time initialize it to the values of object C1.

- Another form of this statement is copyconst C3=C1
- The process of initialising through a copy constructor is known as "copy initialisation".

imp

* CONSTRUCTOR OVERLOADING :-

- Process of defining more than one constructors in a single class is known as constructor overloading.

- If we are creating n constructors in a class then we have to initialise minimum n objects by using each constructor at least once.

```
#include <iostream.h>
#include <conio.h>
class code
{
    int i;
    public:
      code ()
      {
          i = 10;
      }
      code (int a)
      {
```

```
        i = a;
    }
    code (code &x)
    {
        i = x.i;
    }
};
void show()
{
    cout << i;
}
};
void main()
{
    code C1;
    code C2(30);
    code C3;
    code C4(50);
    code C5(C3);
    code C6(C4);
    C1.show();
    C2.show();
    C3.show();
    C6.show();
    getch();
}
```

H.W :-

Diff b/w OOP & PO

output :-

10
30
10
50

10/8

* Constructors with default arguments :-

- It is possible to define constructors with default arguments.

For eg :- A constructor for class ABC can be defined as -

```
ABC (int a, int b = 10)
{

}
```

- The default value of the argument 'b' is 10. The statement "ABC a1 (40);" assigns 40 to argument a and 10 to argument b.

- However, the statement "ABC a2 (40, 50);" assigns 40 to argument 'a' and 50 to argument 'b'.

- The actual parameter when specified overwrites the default value. Such constructors are known to be constructors with default arguments.

Imp

* Destructors :-

- Destructor is used to destroy the objects that have been created by a constructor.
- Destructor is a member function whose name is same as that of the class but is preceded by the symbol '~'.
- A destructor never takes any argument nor does it return any value.
- It will be invoked implicitly by the compiler upon exit from the program.
- It is a good practice to declare destructors in a program since it releases memory space for future use.
- Syntax of destructor for a class name test, the destructor will be as follows :-

```
~Test()
{
  Cout<<" I am destructor";
}
```

- Program for destructor :-

```
Class Test
{
  public:
  Test()
  {
    Cout<<"I am constructor";
  }
```

```
    void show()
    {
        Cout << " Lab assignment to be submitted on 18|8|18";
    }
    ~Test()
    {
        Cout << " I am destructor";
    }
};
int main()
{
    clrscr(); → if it is not written then destructor will print its work.
    Test t1, t2;
    t1.show();
    t2.show();
    return 0;
    getch();
}
```

Program:-

```
#include<iostream.h>
#include<conio.h>
class alpha
{
    int count=0;
    public:
    alpha()
    {
        count++;
        Cout << "No. of object created" << count;
    }
```

```
~alpha()
{
    cout<<"No. of object destroyed"<< count;
    count--;
}
};
int main()
{
    alpha a1, a2, a3, a4;
    cout<<"Hello all";
    getch();
    return 0;
}
```

→ Output:-

```
No. of object created 1
No. of object created 2
No. of object created 3
No. of object created 4
Hello all
No. of object destroyed 4
No. of object destroyed 3
No. of object destroyed 2
No. of object destroyed 1
```

→ this executes after exit of program & running again the program

- Objects are destroyed in the reverse order of creation.

Static Keyword

Static functions

Static data members

- Static Member Functions :-

⇒ A member function that is declared static has the following properties:-

a) A static function can have access to only other static members declared in the same class.

b) A static member function is called using the class name instead of its objects as follows:-

class_name :: fn_name ( );

```
class Late
{
    public:
static void test ( )
{
    cout << "I am static member function";
}
 void demo ( )
{
 cout << "Reminder, Lab assignment last date 18/8/18";
}
void main ( )
{
    clrscr ( );
    late :: test ( );
    late l1;
    l1. demo ( );
    getch ( );
}
```

Imp

• Static Data Members

⇒ A data member of a class can be qualified as static.

⇒ A static member variable has certain special characteristics. These are:-

a) It is initialized to zero (0) when the first object of its class is created. No other initialization is permitted.

b) Only one copy of that member is created for the entire class and is shared by all the objects of that class no matter how many objects are created.

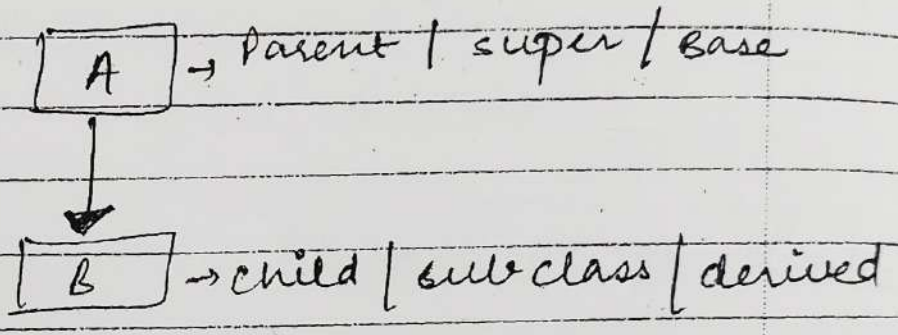
⇒ Static variables are normally used to maintain values that are common to the entire class.

```
class item
{
    static int count;
    int number;
public:
    void getdata(int a)
    {
        number = a;
        count++;
    }
void getcount()
    {
        cout << " count : " << count;
    }
};
```

20/8

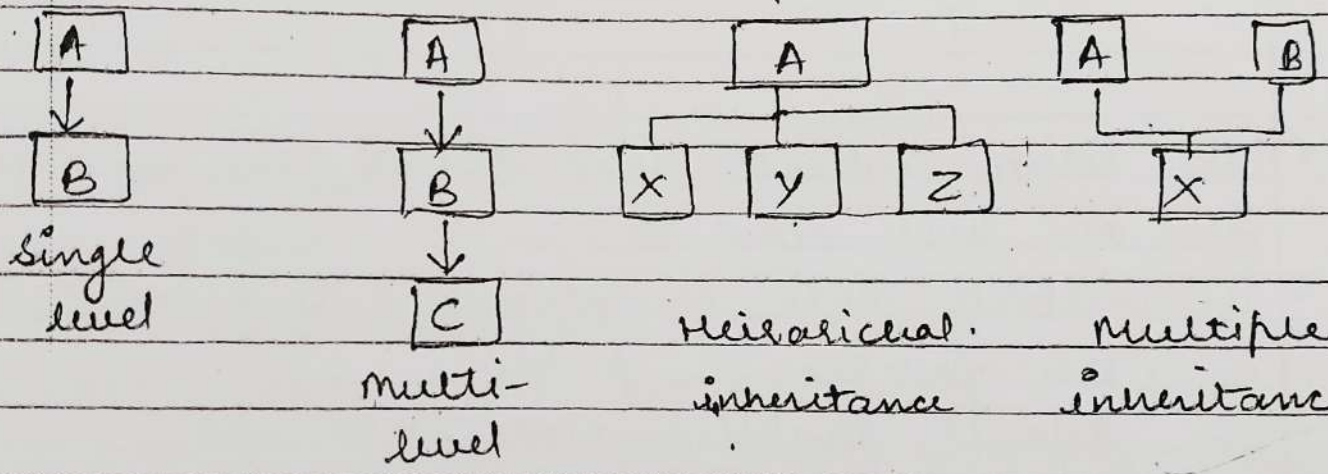
INHERITANCE

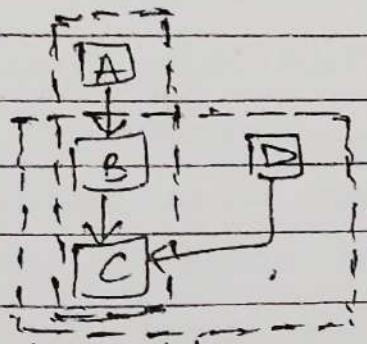
- The mechanism of deriving a new class from an old one is called inheritance.
- The old class is refer to as a base class and a parent class and the new one is called derived class or sub-class.



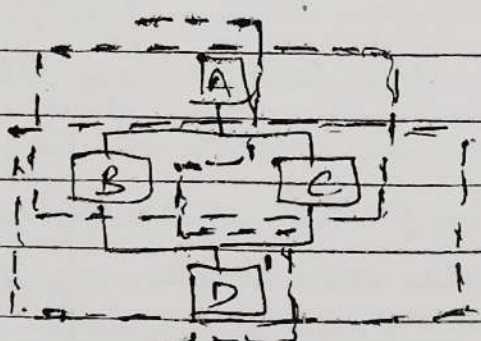
- Inheritance is used to support the concept of reusability.
- The derived class inherits some or all the traits from the base class.
- A class can also inherit properties from more than one class or from more than one level.

- A derived class with only one base class is known as "Single level inheritance".
- A derived class with several base classes is called "multiple inheritance."
- The traits of one class may be inherited by more than one class, this is known as "heirarichal inheritance"
- The mechanism of deriving a class from another derived class is known as "multi-level inheritance."
- The combination of multi-level, multiple or heirarichal is known as "hybrid inheritance."





Hybrid Inheritance



Hybrid inheritance

Page No. Date

Imp

Pg-210, 211, 212, 213

Base class Visibility	Derived class visibility: Public Derivation	Private Derivation	Protected Derivation
private	X	X	X
protected	protected	private	protected
public	public	private	protected

* Types of INHERITANCE :-

1) SINGLE LEVEL :-

```
# include <iostream.h>
# include <conio.h>
class A
{
    int a, b, c;
    public:
    void sum()
    {
        cout << " Enter two values: ";
        cin >> a >> b;
        c = a + b;
        cout << " sum is " << c;
    }
};
class B : public A
{
    int study;
    public:
```

```
        void study()
        {
            cout<<" studying inheritance in c++";
        }
};
void main()
{
        classa();
        B x;
        x.sum();
        x.study();
        getch();
}
```

2.) MULTI-LEVEL :-

```
#include<iostream.h>
#include<conio.h>
class AA
{
    public:
    void show()
    {
        cout<<" inheritance"<<"\n";
    }
};
class BB: public AA
{
    public:
    void display()
    {
```

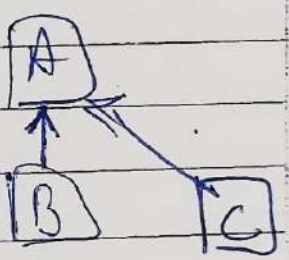
```
        cout<<" Multi-level inheritance"<<"\n";
}
class CC : public BB
{
        public:
        void test()
        {
                cout<<" Test of constructors tomorrow";
        }
};
void main()
{
        clrscr();
        CC x;
        x.show();
        x.display();
        x.test();
        getch();
}
```

Output :-
Inheritance
Multi-level inheritance
Test of Constructor tomorrow

3) HEIRARICHAL INHERITANCE :-

```
#include<iostream.h>
#include<conio.h>
class AAA
{
        public:
        void hello()
        {
                cout<<" hello bsc"<<"\n";
        }
};
```

```
class BBB : public AAA
{
    public:
    void bye()
    {
        cout<<" bye"<<"\n";
    }
};
class CCC: public AAA
{
    public:
    void hie()
    {
        cout<<" hie"<<"\n";
    }
};
void main()
{
    clrscr();
    BBB x;
    CCC y;
    x.hello();
    x.bye();
    y.hello();
    y.hie();
    getch();
}
```



Output:

```
hello BSC
bye
hello BSC
hie
```

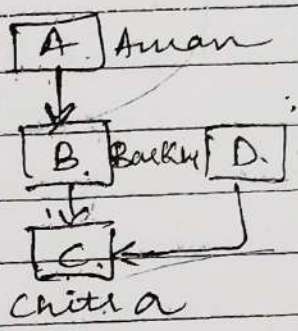
4. MULTIPLE INHERITANCE

```
#include<iostream.h>
#include<conio.h>
class A
{
    public:
    void display()
    {
        cout<<" Multiple inheritance"<<"\n";
    }
};
class B;
{
    public:
    void show()
    {
        cout<<" Example"<<"\n";
    }
};
class C: public A, public B
{
    public:
    void get()
    {
        cout<<" Inheritance done"<<"\n";
    }
};
void main()
{
    clrscr();
```

```
c x;
x.display();
x.show();
x.get();
getch();
}
```

Output:
Multiple Inheritance
Example
Inheritance done

5) HYBRID INHERITANCE




```
#include<iostream.h>
#include<conio.h>
class Aman
{
    int a, b, c;
    public:
    void study()
    {
        cout<<" studying "<<"\n";
    }
};
class Barkha : public Aman
{
    public:
    void test()
    {
        cout<<" Test "<<"\n";
    }
};
class Devesh ~~Chetna : public Barkha~~
{
    public:
```

```
        void show()
        {
            cout << " show" << "\n";
        }
};
class Devesh : public Barkha, public Dinesh
{
    public:
    void take()
    {
        cout << " take" << "\n";
    }
};
void main()
{
    clrscr();
    Chitra x;
    x.study();
    x.test();
    x.show();
    x.take();
    getch();
}
```

Output :-

studying
Test
show
Take

Multi-level inheritance program :-

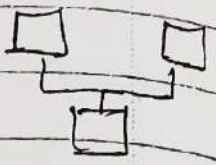
```
#include <iostream.h>
#include <conio.h>
class A
{
    public:
    A(int a)
    {
        cout<<" Hie" << a;
    }
    void recall()
    {
        cout<<" Nisha";
    }
};
```

```
class B : public A
{
    public:
    B(int m, int n) : A(m)
    {
        cout<<" Hello "<<m<<n;
    }
    void revise()
    {
        cout<<" Nitish";
    }
};
class C : public B, public B
{
    public:
    C(int x, int y, int z) : B(y, z)
    {
        cout<<" Hey "<<x<<y<<z;
    }
    void study()
    {
        cout<<"Aman";
    }
};
void main()
{
    clrscr();
    C h(2, 3, 4);
    h.recall();
    h.revise();
    h.study();
    getch();
}
```

Output:

```
Hie 3
Hello 3 4
Hey 2 3 4
Nisha
Nitish
Aman
```

* Multiple Inheritance Program :-



```
#include <iostream.h>
#include <conio.h>
class DAU
{
    public:
    DAU(int a)
    {
        cout <<" DAU" << a;
    }
    void show()
    {
        cout<<" show" ;
    }
};
class MDU
{
    public:
    MDU(int m, int n)
    {
        cout <<" MDU" << m << n;
    }
    void display()
    {
        cout <<" display";
    }
};
class BSC : public DAU, public MDU
{
    public:
    BSC(int x, int y, int z): DAU(x), MDU(y,z)
    {
```

```
        cout<<" BSC " << x << y << z;
    }
    void assignment ()
    {
        cout << " assignment ";
    }
};
void main()
{
    clrscr();
    BSC b(4, 5, 6);
    b.show();
    b.display();
    b.assignment();
    getch();
}
```

Output :-

```
DAU 4
MDU 5 6
BSC 4 5 6
show
display
assignment
```

* CONSTRUCTORS IN DERIVED CLASS :-

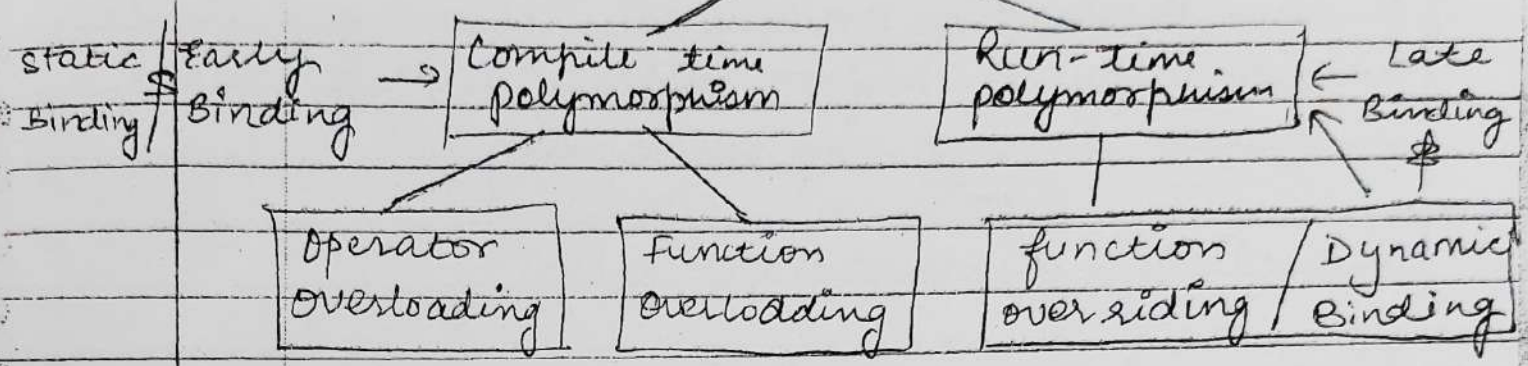
- If we create constructors in both base class & derived class then base class constructor is invoked before derived class constructor.
- As long as no base class constructor takes any arguments, the derived class need not have a constructor function.
- However, if any base class contains a constructor with one or more arguments then it is mandatory for derived class to have a constructor & pass a arguments to the base class constructor. The general form of defining such derived constructor is as follows:

~~Derivedconstr(arg1, arg2, ... argN) : base~~

```
derivedconstr (arg1, arg2, ..... argN, argD):
    base1 (arg1),
    base2 (arg2),
      :
      :
    baseN (argN),
{
    body of derived constr
}
```

- The header line of derived constructor function contains two parts separated by a colon, the first part provides the declaration of the arguments that are passed to the derived constructor & the second part lists the function call to the base class constructor

POLYMORPHISM



- Polymorphism is one of the crucial features of OOP.
- Polymorphism means one name, multiple forms. (ability to take more than one form).
- Polymorphism can be classified as:

1) Compile-time
2) Run-time

1) COMPILE-TIME POLYMORPHISM:

- An object is bound to its function call at compile time.
- It is also known as early binding or static binding.
- In compile-time polymorphism, the compiler is able to select the appropriate function for a particular call at the compile-time.
- Compile-time polymorphism can be classified as:

a) operator overloading
b) function overloading

a) Operator Overloading - The process of making an operator to exhibit different behaviours in different instances

is known as operator overloading.

b) Function Overloading - The process of using a single function name to perform different type of tasks is known as function overloading.

⇒ Run-Time Polymorphism :-

- Binding of an appropriate function for a particular function call is not done by the compiler and is done at run-time.
- It is also known as late binding or dynamic binding.
- Dynamic Binding requires the use of pointers to the objects.

* FUNCTION - OVERLOADING :-

- By Changing the number of arguments -

```
#include (iostream.h)
#include (conio.h)
Class BSC
{
    public:
    void perimeter (int s)
    {
        int x,p;
        x = s;
        p = 4 x x;
        Cout << " perimeter of Square is: " << p;
    }
```

```
void perimeter (int l, int b)
{
        int m, n, y;
        m = l;
        n = b;
        y = 2(m+n);
    Cout << "perimeter of rect. is:" << y;
    }
};
void main()
{
    clrscr();
    BSC b1;
    b1. perimeter (5);
    b1. perimeter (3, 5);
    getch();
}
```

- By changing the type of arguments:-

```
#include <iostream.h>
#include <conio.h>
class BSC
{
    public:
    void area (int s)
    {
        int a
        a = s x s;
        cout << "area of square:" << a;
    }
```

```
void area(int l, int b)
{
    int a;
    a = l * b;
    cout << "area of rectangle :" << a;
}
void area(float r)
{
    int a;
    a = 3.14 * r * r
    cout << "area of circle :" << a;
}
};
void main()
{
    clrscr();
    BSC x;
    x.area(5);
    x.area(3, 4);
    x.area(7.2);
    getch();
}
```

Q- WAP for copy constructor

```
#include <iostream.h>
#include <conio.h>
class copy
{
    public:
```

```
void copy(int x, int y)
{
    int p;
    m = x;
    n = y;
    p = m+n;
    cout << "sum:" << p;
}
void copy (copy &s)
{
    m = s.m;
    n = s.n;
}
void show()
{
    cout << m << n;
}
void main()
{
    clrscr();
    copy z; copy z(3,4);
    z.show(); copy z1(z);
    getch();
    copy z1(3,4);
    copy z2(z1);
    z1.show();
    z2.show();
    getch();
}
```

11/9

OPERATOR OVERLOADING

- The process of making an operator to exhibit different behaviour in different instances is known as operator overloading.

→ For eg - C++ permits us to add two variables of user defined types with the same syntax that is applied to the basic types.

$$3 + 4 = 7$$

"Hello" + "All" = Hello all

- Operator overloading provides a flexible option for the creation of new definitions of most of the C++ operator.
- We can overload all the c++ operators except the following :-

1.) class members access operators (.) (Dot operator)
2.) scope resolution operator (::)
3.) Size operator (size of)
4.) conditional operator (?:)

- Defining Operator Overloading-

To define an additional task to an operator we must specify what it means in relation to a class to which the operator is applied. This is done with the help of a special function called operator function. The general form of an operator function is as follows :-

return type class :: operator op (arguments)

{

function body // task defined

}

→ where return type is the type of value returned by the specified operation and 'op' is the operator being overloaded.

• 'op' is preceded by the keyword "operator".

• The process of overloading involves the following steps.

step 1: Create a class that defines the data type that is to be used in the overloading operation.

Step 2: Declare the operator function in the public part of the class.

Step 3: Define the operator function to implement the required operations.

* FUNCTION OVER-RIDING :-

- Having same name function with same number and type of arguments in base class and derived class.
- When we use the same function name in both the base & derived classes, the function in base class is declared as virtual using the keyword "VIRTUAL" preceding its normal declaration.
- When a function is made virtual C++ determines which function to use at run-time based on the type of object pointed to by the base pointer rather than the type of pointer.

```
#include<iostream.h>
#include<conio.h>
class base
{
    public:
    void display()
    {
        cout<<"Display base";
    }
    virtual void show()                    //virtual function
    {
        cout<<" show base";
    }
};
class derived : public base
{
    public:
    void display()
    {
        cout<<" Display derived";
    }
    void show()
    {
        cout<<" show derived";
    }
};
int main()
{
    base b;
    derived d;
    base *b1;
```

```
    bl = &b;
    bl -> display();
    bl -> show();
    bl = &d;
    bl -> display();
    bl -> show();
    getch();
    return 0;
}
```

Output :-

```
Display Base
Show Base
Display Base   <- func^n overriding
Show derived
```

] - run-time polymorphism

Thankyou For Reading

Like,Share and Comment

https://www.linkedin.com/in/deepa-chaurasia-notes