

## OBJECT ORIENTED PROGRAMMING:

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

Object oriented programming allows the programming to be closer to real world and thereby making it less complex.

### Classes And Objects:

A class is a way to bind the data describing entity and its associated functions together.

For instance, consider a user having characteristics username and password and some of its associated operations are sign up, sign in and logout.

Class is just a template, which declares and defines characteristics and behaviour, hence we need to declare objects of the class for it to be usable. In other words class represents a group of similar objects.

### Data members and member function:

Data ~~in~~ members are the data-type properties that describe the characteristics of the class.

Member functions are the set of operation that

may be applied to objects of class, i.e., the coherent the behavioural aspect of the object.

Syntax for defining class:

class class\_name {

Access Modifiers:

Data members

Member functions

}

;

The class body contains the declaration of members (data & functions).

Declaring Objects of a class:

We can declare objects of class either statically or dynamically just as we declare variable of primitive data types.

(i) Statically:

class\_name object\_name; ] Syntax

(ii) Dynamically:

class\_name \* object\_name = new class\_name ] Syntax

Example code for declaring statically:

```
class Student {  
public:  
    int rollno;  
    char name[20];  
};  
int main () {  
    Student S1; // Declaring Objects.  
    Student S2;  
}
```

Example code for declaring dynamically:

```
class Student {  
public:  
    int rollno;  
    char names[20];  
};  
int main () {
```

Student \*S1 = new Student // Declaring object  
// of type Student  
// dynamically.

## Access Modifiers:

- The **private** members can be accessed only from within the class. These members are hidden from the outside world. Hence they can be used only by members functions of the class in which it is declared.
- The **public** members can be accessed outside the class ~~only~~ also. They can be directly accessible by any function, whether member function of class or non-member function.
- By default members of class are **private**.

Ex: ~~#include<iostream.h>~~  
using namespace std;

```
class Student { }           // Declaring class
public:                   // Access modifiers
    int rollno;
    int age;
};

int main() {
```

```
    Student S1;           // Create objects statically
    Student S2;
```

Student S3, S4, S5;

// can be declared like this also

S1. age = 24;

// Assigning values

S1. rollno = 101;

cout << &S1. age << endl; // Pointing value

cout << S1. rollno << endl;

Student \*S6 = new Student; // Create object dynamically

(\*S6). age = 23; // Assigning value

(\*S6). rollno = 104;

// Can also be done in a way given below:

S6 → age = 23;

S6 → rollno = 104;

}

### Getters And Setters:

The private members of class are not accessible outside the class, although sometimes there is necessity to provide access to even private members, in these cases we need to create functions called getters and setters. They are also called as accessor and mutator function respectively.

Example Code:

```
class Student {
```

```
    int rollno;  
    char name[20];  
    float marks;  
    char grade;
```

```
public :
```

```
    int getRollno() {
```

```
        return rollno; }
```

```
    int getMarks() {
```

```
        return marks;  
    }
```

```
    void setGrade() {
```

```
        if (marks > 90) grade = 'A';  
        else if (marks > 80) grade = 'B';  
        else if (marks > 70) grade = 'C';  
        else if (marks > 60) grade = 'D';  
        else grade = 'E';  
    }
```

getRollno() & getMarks are getter & setGrade is setter.

If we need to define member functions outside class:

```
int Student :: getMarks () {  
    return marks;  
}
```

```
int Student :: getRollno () {  
    return rollno;  
}
```

We can access member functions in similar manner via an object of class Student and <sup>using</sup> dot operator.

## Constructor:

A constructor in a class is means of initialisation or creating objects of a class. A constructor allocates memory to the data members when an ~~an~~ object is created. It may also initialise the object with legal initial value.

## Characteristics of Constructor:

- Constructor is a member function of a class and has same name as that of class.
- Constructor functions are invoked ~~at~~ automatically when the objects are created.
- Constructor functions obeys the usual access rules. That is, private constructor are available only for member functions, however, public are available for all functions.
- Constructor has no return type, not even void.

### Default Constructor:

A constructor that accepts no parameter is called default constructor. The compiler automatically supplies a default constructor implicitly. This constructor is called public member of class.

### Example code:

```
class Sum {  
    int a, b;  
public:  
    int getsum(){  
        return a+b;  
    }  
};  
int main(){  
    Sum obj; // Implicit default constructor invoked  
    }  
}
```

- Creating your own default constructor:

When a user-defined default constructor is created, the compiler's implicit default constructor is overshadowed.

Example Code:

```
class Sum {
```

```
    int a, b;
```

```
public:
```

```
    Sum() {
```

// User-defined default constructor

```
    cout << "Constructor invoked: ";
```

```
    a = 10;
```

```
    b = 20;
```

```
}
```

```
    int getSum() {
```

```
        return a+b;
```

```
}
```

```
}
```

```
int main() {
```

```
    Sum obj; // Explicitly defined default constructor called
```

```
}
```

Output = Constructor invoked:

## Parameterized Constructor:

The constructor that can take arguments are called parameterized constructor.

### Example Code:

```
class Sum{
```

```
    int a,b;
```

```
    public:
```

```
        Sum ( int num1, int num2 ) { } //Parameterised  
        a = num1 ; // constructor  
        b = num2 ; //
```

```
    int } getSum () {
```

```
        return a+b; }
```

```
};
```

```
int main ()
```

```
    Sum obj(4,2); //Parameterised constructor invoked
```

```
}
```

Declaring the constructor with arguments hides the default constructor. Hence, the object declaring statement such as Person obj;  
may not work.

- Any number of constructors may be declared and the constructor that would be invoked would be according to the parameters.

Eg:- Student S1; // constructor 1 will be called

Student S2(101); // constructor 2 will be called

Student S3(20,102); // constructor 3 will be called

Student S4(S3); // copy constructor

$S_1 = S_2$  // copy assignment operator

Student S5 = S4; // copy constructor

Copy constructor is inbuilt ~~with~~.

## Destructors:

A destructor as the name suggest is used to destroy the objects that have been created by a constructor. A destructor is also a member function whose name is the same as the class name but preceded by a tilde ('~').

For eg, destructor to class Sum is `~Sum()`.

- A destructor takes no argument and has no return type.
- It is automatically called by compiler when object is destroyed.
- But for dynamically declared object it does not call itself & we have to use keyword `delete object-name;`
- As soon as object goes out of scope, destructor is called and object is destroyed releasing its occupied memory.
- Destructors are invoked in reverse order as ~~destructors~~<sup>constructors</sup> are invoked.

Example code:

```
class Sum {
```

```
    int a, b;
```

```
public :
```

```
    Sum (int num1, int num2) {
```

```
        cout << "Constructor at work" ; }
```

```
    ~Sum () {
```

```
        cout << "Destructor at work" ; }
```

```
    int getSum () {
```

```
        return a + b;
```

```
    }
```

```
int main () {
```

```
    Sum obj (4, 6);
```

```
}
```

Output = Constructor at work

Destructor at work

## ※ this keyword:

C++ uses a unique keyword called `this` to represent an object that invokes member function. `this` is a pointer to the object for which the function was called. For example, the function call `obj.getSum()` will set the pointer `this` to the address of the object `obj`. This unique pointer is automatically passed to a member function when it is called. The pointer `this` acts as an implicit argument to all the member functions.

Example code :

```
class Sum{  
    int a, b;
```

Public:

```
    Sum (int a, int b){
```

this → a = a;

this → b = b;

}

```
    int getSum(){
```

return a + b;

}

};

In the constructor of the `Sum` class, since the data members and data members have same name, this keyword is used to differentiate between the

APCO  
Date :  
Page :  
two. Here this  $\rightarrow$  a refers to data member a of object obj.

Ques: Make a class named fraction having a fraction number. Also closure operations.

class Fraction {

private :

```
int numerator;  
int denominator;
```

public :

fraction( int numerator, int denominator ) {

    this  $\rightarrow$  numerator = numerator;

    this  $\rightarrow$  denominator = denominator;

}

void print () {

    cout  $\ll$  numerator  $\ll$  " / "  $\ll$  denominator " ;  
    T  
     $\rightarrow$  or compass const fraction & so that no copying  
    takes place,

void add ( fraction f2 ) {

    int lcm = denominator \* f2.denominator;

```
int x = lcm / denominator;
```

```
int y = lcm / f2.denominator;
```

```
int num = x * numerator + (y * f2.numerator);
```

numerator = num;      // Updating sum in  
denominator = lcm;      f2 only.

```
int gcd = 1;
```

Find  
HCF  
int j = min(numerator, denominator);

for (int i = 1; i <= j; i++)

if ( numerator % i == 0 && denominator % i == 0 )

gcd = i;

}

numerator = numerator / gcd;

denominator = denominator / gcd;

}

```
int main(){
```

```
    Fraction f1(10,2);
```

```
    Fraction f2(15,4);
```

```
    f1.add(f2);
```

```
    f1.print();
```

```
    f2.print();
```

```
}
```

Output: 
$$\begin{array}{r} 35/4 \\ 15/4 \\ \hline \end{array}$$

eg: Class with characters also:

```
#include <iostream>
using namespace std;
#include "Student.hpp" //Adding classfile
```

```
int main()
```

```
char name[] = "abcd";
```

```
Student s1(20, name);
```

```
s1.display();
```

```
name[3] = 'e';
```

```
Student s2(24, name);
```

```
s2.display();
```

```
}
```

Class on next page

class Student {

int age;  
char \*name;

public :

Student ( int age, char \*name ) {

this → age = age;

Way: 1

// Shallow copy

this → name = name;

Way: 2

// deep copy

this → name = new char [ strlen(name) + 1 ];

strcpy ( this → name, name );

}

void display () {

cout << name << " " << age << endl;

}

};

We have to use either method for copying. Output would be as follows:

Way: 1

Output = 

a b c d	20
a b c e	24
<u>a b c e</u>	20

Way: 2

Output = 

a b c d	20
a b c e	24
<u>a b c d</u>	20

In way 1 all three name,  $s_1$   $s_2$  were pointing to same array. So when  $a b c d$  changes to  $a b c e$  it reflects not only in  $s_2$  but  $s_1$  also.

In way 2 a new ~~pointer~~<sup>array</sup> is created that ~~is~~ a pointer in  $s_1$  is pointing. So any change in  $s_2$  or name does not effect it.

## Copy Constructor :

eg: Using previous class Student.cpp :-

with ~~should be~~ copy

Just make name as public

```
#include <iostream>
using namespace std;
#include "Student.cpp"
```

```
int main () {
```

```
char name [] = "abcd";
```

```
Student S1 (20, name);
```

```
S1.display ();
```

```
Student S2 (S1); // Default Copy Constructor
```

```
S2.name [0] = 'x';
```

```
S1.display ();
```

```
S2.display ();
```

Output:

20 abcd

20 Xbcd

20 Xbcd

\* \* Here even we have done deep copying but Xbcd is only printed for S1 as here when default constructor is called S2 also points to same memory. So changes in one changes other.

So indirectly default constructor goes for shallow copying instead of deep copying.

So we can create our own copy constructor in that class which would be:

Student ( Student & s ) {

    this → age = s . age

    this → name = new char [ strlen ( s . name ) + 1 ] ;

    strcpy ( this → name , s . name ) ;

3.

After writing this in class no default constructor

Here argument of copy constructor needs copy constructor  
so cast to pass by value.  
So pass it by reference i.e.

Student ( Student & s ) {

    But we want to make it ~~as~~ i.e. don't want  
    changes ~~as~~ due to this so make

Student ( Student const & s ) {

INITIALISATION LIST:

- Whenever there are constant variable or reference variables in class we require initialisation list.

class Student {

public:

```
const int roll number;  
int age;  
int & x; // age reference variable
```

[  
Student (int x, int age) : rollNumber(x), age(age),  
x (this->age) {  
};

Here while writing code as soon as compiler sees  
written any object declaration generates error  
if ([ ]) bracketed statement is not there.

As this means initialisation & so putting garbage  
values to all elements in class. Putting garbage  
value in const & or & variable generates  
error and so we must use statement,

## Constant Functions:

- If for a class const and object is created then it can only call constant functions.
- These are the functions that does not change any property of current object.
- To make a function constant add `const` keyword after function name.

Ques:- What will be output:

class Student {

public:

int rollnumber;

int age;

}

int main()

Student s1;

Student const s2 = s1;

s1. rollnumber = 10;

s1. age = 20;

cout << s2. rollnumber << " " << s2. age;

Output = Garbage Garbage

Ques:- What is the Output:

class Student {

int rollnumber;

public:

int age;

Student ( int x ) {

rollnumber = x;

int getRollNumber () {

return rollnumber;

}

int main () {

Student s1 ( 101 );

s1. age = 20;

Student const s2 = s1;

cout << s2. getRollNumber ();

}

Output = Error [ As const. only calling const. function ]

## Static Properties of Class:

eg:- Class Student?

Public:  
int rollnumber;  
int age;  
static int totalStudents;  
};

Here in the given class if we create different Student S1, S2... as objects of class must have different rollnumbers and may have ~~same~~ different age but total number of students must be same. Such kind of property of class which must be same for all objects of class are declared as static by using keyword static.

Now, if we make different objects S1, S2 they have age and rollnumber only. Static variable such as total students are only mentioned in class & copies of these are not formed in different objects.  
We access static data members by:

int Student::totalStudents = 10 ;

cout << Student::totalStudents << endl;

Also can be accessed by Student::totalStudents ;

If we need to add total no of student acc to total objects made we can ~~as~~ take help from constructor. For eg:

class Student {

public:

int rollnumber;  
int age;

Students () {

totalStudent++;  
}

So, after ~~each~~<sup>delevery</sup> object  
total student value  
increases by 1

int getRollnumber () {

return rollnumber; }

static int getTotalStudents () {

return totalStudents;  
}

Static function,  
call also be  
made

};

- To access static function we use same scope resolution operator.

Eg:-

Student:: getTotalStudents();

- We declare static those data members and functions which we want same for all objects.
- Static functions may only access static members.  
i.e static data members or static function.
- Static function does not have any this →  
pointer access.

## Operator Overloading:

In previously created fraction class we have one function:

Fraction add (Fraction const& f2) {  
    ~~3~~

If we want to overload ~~the + operator~~ we can rename it as

Fraction operator + (Fraction const& f2) {  
    ~~3~~

So,  $f1.add(f2) = f1 + f2$

eg:- ~~Overload Operator ++ (Pre increment)~~

If there are both host & pre increment operator overloading then

Function Overload  
Fraction operator ++ () { || Pre

Fraction operator ++ (int) { || Post

eg:- Class Dynamic array:

class DynamicArray {

int \* data;  
int next\_index;  
int capacity;

public:

DynamicArray()

|| (Constructor): initial values

data = new int [5];  
next\_index = 0;  
capacity = 5;  
}

DynamicArray (DynamicArray const & d) { || (Copy  
constructor  
for deep  
copy)

this → data = new int [d.capacity];

for (int i = 0; i < d.next\_index; i++) {

this → data[i] = d.data[i];

}

this → next\_index = d.next\_index;

this → capacity = d.capacity;

}

// Copy Assignment Operator

void operator= (DynamicArray const& d) {

Some line of  
copy construct

} void add (int element) {

// Adding element  
to dynamic array

if (next\_index == capacity) {

int \* newData = new int [ 2 \* capacity];

for (int i=0; i < capacity; i++) {

newData[i] = data[i];

delete [] data;

data = newData;

capacity = capacity \* 2;

}

data [next\_index] = element;

next\_index ++;

}

```
int get (int i) {  
    if (i < next_index) {  
        return data[i];  
    }  
    else {  
        return;  
    }  
}  
  
void add (int i, int element) {  
    if (i < next_index) {  
        data[i] = element;  
    }  
    else if (i == next_index) {  
        add (element);  
    }  
    else {  
        return;  
    }  
}  
  
void print () {  
    for (int i = 0; i < next_index; i++) {  
        cout << data[i] << " ";  
    }  
}
```

\* Remedy oches after uestions

# OOPTs CONT. :

## Abstraction And Encapsulation:

- Clubbing data and functions of any entity is called encapsulation. Class helps in achieving encapsulation.
- Abstraction is hiding unnecessary details from outside world.

## INHERITANCE:

- Inheritance is the capability of one class of things to derive capabilities or properties from another class.
- Access modifier, protected, all the data members that are protected can only be used by child class.
- Inheritance allows the addition of additional features to an existing class without modifying it. One can derive a new class (sub class) from an existing one and add new features to it.
- Only public and protected are accessible to child class to an parent class.

- Syntax:

```
class Vehicle {
```

private:

```
int maxspeed;
```

protected:

```
int numtyres;
```

public:

```
string colour;
```

```
};
```

Now, if we want to inherit class car from vehicle  
we would write syntax as:

```
class car : access-specifier Vehicle {}
```

This access-specifier may be public, private or  
protected.

(i) If access-modifier is public i.e. inheriting publicly

private of vehicle  $\rightarrow$  X never inherited  
protected of vehicle  $\rightarrow$  protected of car  
Public of vehicle  $\rightarrow$  public of car.

(ii) If access-modifier is protected:

private  $\rightarrow$  Not accessible

public & protected of vehicle class becomes  
protected of car class.

(iii) If access-modifier is private's

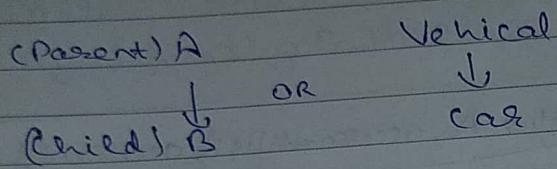
private  $\rightarrow$  Not accessible.

public and protected of vehicle class becomes  
private of car class.

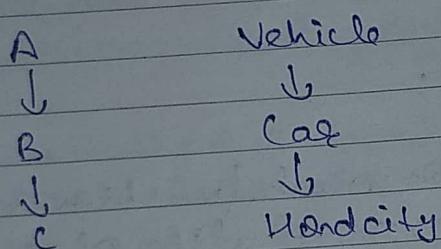
- If access-modifier is not specified then by default it is taken as private.
- If object of ~~car~~ class car is created then first constructor of parent class i.e. public is called and then constructor of car class is called.
- In destructor call is just opposite. First of child class then of parent class.

## Types of Inheritance:

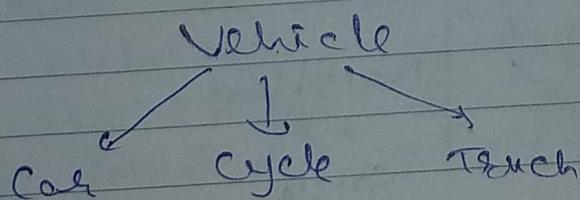
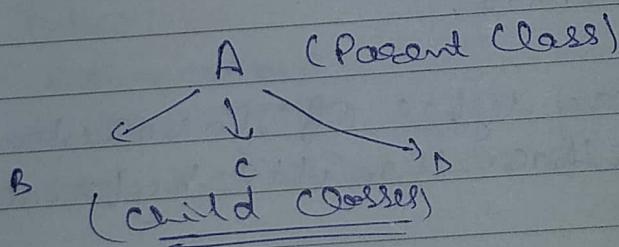
### (i) Single Inheritance:



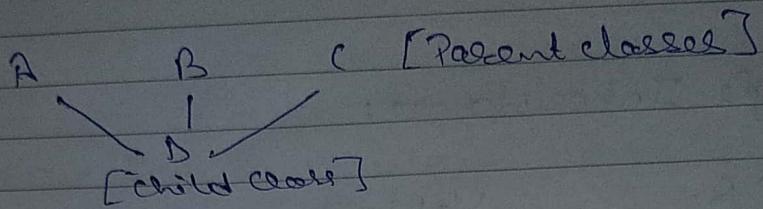
## (ii) multilevel Inheritance:



### (iii) Hierarchical instance:



## (iv) Multiple Inheritance:



class D : access modifier A, access modifier B, access modifier C

If two function let void print C() be there in all A, B, C then

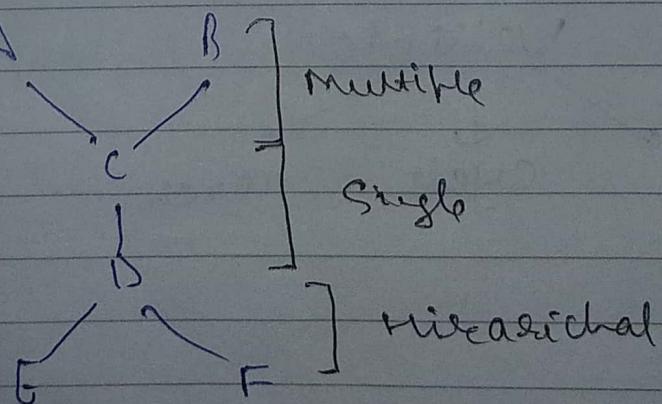
④ A:: parent();

we have to mention which function is called using ~~as~~ scope resolution operator.

## (v) Hybrid Inheritance:

If two or more types of Inheritance are used together then it is hybrid inheritance.

Eg:-



## POLYMORPHISM:

Polymorphism is a property by which the same message can be sent to objects of several different classes, and each object can respond in different way depending on its class.

### (i) Compile Time Polymorphism:

- Compile time polymorphism may be achieved by function overloading and operator overloading.
- Other method is function overriding; in this if parent class and child class both have function with same name.

e.g:- class vehicle {  
public:

    string colour;  
    void print();  
    cout << "vehicle" << endl; } };

class car : public vehicle {

    public:

    void print();  
    cout << "car" << endl; } };

int main() {

    Vehicle v;  
    Car c;

    v.print();      // Prints vehicle

    c.print();      // Prints car

    Vehicle \*v1 = new Vehicle;

    Vehicle \*v2;

    v2 = &c;

    v1 → print();      // Prints vehicle

    v2 → print();      // Prints vehicle

}

\*\* Here, via v2 only those properties in C  
are accessible which are there in  
Vehicle class also.

### (ii) Run Time Polymorphism :

- In previous case by  $V_2 \rightarrow \text{Point C}$ , vehicle is pointed. If we want car to be pointed we make use of virtual functions.
- Virtual functions are those functions which are present in base class and they are overridden in derived class. We add keyword virtual before function name.

### VIRTUAL FUNCTION AND ABSTRACT CLASS:

- Pure virtual function is one which does not have any definition.  
eg:- virtual void print() = 0;
- Any class that contains pure virtual functions are called abstract class.
- We cannot create object of abstract class.
- For any class to be inherited from abstract class have to either itself also become abstract class or implement all the pure virtual functions.

## FRINED FUNCTIONS AND CLASSES:

- Private members of a class are not accessible outside. But if function of other class wants to access private member of other class, this can be done by making that function friend function of that class. But definition of that function then is given outside the class.

eg :- #include <iostream>  
using namespace std;

class Bus {

public:

void print();  
};

class Truck {

private:

int x;

protected:

int y;

public:

int z;

friend void Bus:: print();

};

```
void Bus::print () {
```

```
    Truck t;
```

```
    t.x = 10;
```

```
    t.y = 20;
```

```
    cout << t.x << " " << t.y << endl;
```

```
}
```

```
int main () {
```

```
    Bus b;
```

```
    b.print ();
```

```
}
```

If we want to make friend a function which is not present in ~~in class~~ any class then we ~~to~~ first declare it above that class then inside class we write

friend void function name ();

then after class we define this function.

• These friend functions cannot use this  $\rightarrow$  pointer.

• We can put these friend function in any access modifier it does not cause any effect.

If we have two classes class Bus & class Truck and class Bus have large no. of functions which we have to make friend function then either we can do it by making separately each function as friend or we can make whole class as friend by writing in truck class the statement

friend class Bus;

( ) 16/07/2021

16/07/2021

16/07/2021

16/07/2021

16/07/2021

16/07/2021

16/07/2021