

**What is object oriented Programming ? Explain difference between OOP and POP.**

### **1.1. Procedure Oriented Programming Vs Object Oriented Programming**

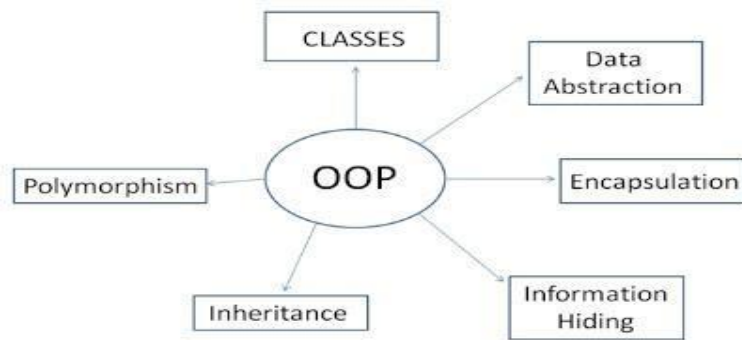
<b>Procedure Oriented Programming</b>	<b>Object Oriented Programming</b>
In POP, program is <b>divided into small parts</b> called <b>functions</b> .	OOP, program is divided into parts called <b>objects</b> .
In POP, <b>Importance</b> is <b>not given to data</b> but to functions as well as sequence of actions to be done.	In OOP, Importance is <b>given to the data</b> rather than procedures or functions because it works as a real world.
POP follows <b>Top Down</b> approach.	OOP follows <b>Bottom Up</b> approach.
POP does <b>not have any access specifier</b> .	OOP has <b>access specifiers named Public, Private, Protected</b> , etc.
In POP, <b>Data can move freely from function to function</b> in the system.	In OOP, <b>objects can move and communicate</b> with each other through member functions.
To <b>add new data and function in POP is not so easy</b> .	OOP <b>provides an easy way to add new data and function</b> .
In POP, Most function <b>uses Global data</b> for sharing that <b>can be accessed freely from function to function</b> .	In OOP, data <b>can not move easily from function to function</b> ,it can be <b>kept public or private</b> so we can <b>control the access of data</b> .
POP does not have any proper way for hiding data so it is <b>less secure</b> .	OOP <b>provides Data Hiding</b> so provides <b>more security</b> .
In POP, <b>Overloading is not possible</b> .	In OOP, <b>overloading is possible</b> in the form of Function Overloading and Operator Overloading.
Example of POP are : C, VB, FORTRAN, Pascal.	Example of OOP are : C++, JAVA, VB.NET, C#.NET.

## **Introduction Of OOP**

- ❑ C++ began as an expanded version of C. The C++ were first invented by **Bjarne Stroustrup in 1980** at **Bell Laboratories** in Murray Hill, New Jersey.
- ❑ This new language was initially called "C with Classes".
- ❑ Object Oriented Programming is a programming in which we design and develop our application or program based of object.
- ❑ **Objects** are instances(variables) of class.

## Basic Concept Of OOP (Features Of OOP):

Que: List out basic concepts of OOP. Explain any two in brief.



### OBJECT:

- ❑ Object is a **collection of number of entities**.
- ❑ Objects **take up space in the memory**. Objects are instances of classes.
- ❑ When a program is executed, the **objects interact by sending messages to one another**.
- ❑ Each object contains data and code to manipulate the data. Objects **can interact without having know details of each others** data or code.

### CLASS:

- ❑ Class is a **collection of objects of similar type**.
- ❑ Class is a **collection of data member and member function**.
- ❑ Objects are **variables of the type class**.
- ❑ Once a class has been defined, we **can create any number of objects** belonging to that class. Eg: grapes, bannans and orange are the member of class fruit.

#### **Example:**

Fruit orange;

- ❑ In the above statement object mango is created which belongs to the class fruit.

**NOTE:** Classes are user define data types.

### DATA ABSTRACTION AND ENCAPSULATION:

- ❑ Combining data and functions into a single unit called class and the process is known as **Encapsulation**.
- ❑ Data encapsulation is an important feature of a class. Class contains both data and functions.
- ❑ **Data is not accessible from the outside world** and only those functions which are present in the class can access the data.
- ❑ The protection of the data from direct access by the program is called **data hiding** or information hiding.
- ❑ Hiding the complexity of program is called **Abstraction** and only essential features are represented. In short we can say that internal working is hidden.

## DYNAMIC BINDING:

- ❑ Refers to **linking of function call with function definition** is called binding and when it takes place at run time called dynamic binding.

## MESSAGE PASSING:

- ❑ The process by which **one object can interact with other object** is called message passing.

## INHERITANCE:

- ❑ It is the process by which object of one class acquires the properties or features of objects of another class.
- ❑ The concept of inheritance provides the idea of **reusability** means we can **add additional features to an existing class without modifying it**.
- ❑ This is possible by deriving a new class from the existing one. The **new class will have the combined features of both the classes**.  
Example: Robin is a part of the class flying bird which is again a part of the class bird.

## POLYMORPHISM:

- ❑ A Greek term means ability to take more than one form. An operation may exhibit different behaviours in different instances.
- ❑ The **behaviour depends upon the types of data used in the operation**.  
**Example:**
  - Operator Overloading
  - Function Overloading

## Benefits Of OOP:

- ❑ Through inheritance, we can **eliminate redundant code** and **extend the use of existing classes** which is not possible in procedure-oriented approach.
- ❑ We can **build programs from the standard working modules that communicate with one another, rather than having to start writing the code from scratch** which happens in procedure-oriented approach. This leads to saving of development time and higher productivity.
- ❑ The principle of **data hiding** helps the programmer to build secure programs that cannot be invaded by code in other parts of the program.
- ❑ It is possible to have **multiple instances of object** to co-exist without any interference.
- ❑ It is possible to **map objects in the problem domain to those in the program**.
- ❑ It is **easy to partition the work** in a project based on **objects**.

- ❑ The **data-centered design** approach enables us to capture more details of a model inimplementable from.
- ❑ Object oriented systems can be **easily upgraded from small to large systems**.
- ❑ **Message passing techniques** for communication between objects makes the interface descriptions with external systems much simpler.
- ❑ **Software complexity can be easily managed**.

## C++ Programming:=

- ❑ C++ is a object oriented Programming Language .
- ❑ It was developed by Bjarne Stroustrup at AT & T Bell Laboratories in New Jersey,USA 1980.

## C++ Basic Program:=

```
#include
<iostream.h>
#include <conio.h>
void main()
{
    cout << "Hello this is
    C++";getch();
}
```

### Header files

- ❑ Header Files are included at the beginning just like in C program.
- ❑ Here **iostream** is a header file which provides us with input & output streams.  
**Header files contained predeclared function libraries**, which can be used by users for their ease.

### main()

- ❑ main() is the function which **holds the executing part of program**.

### cout <<

- ❑ cout<< is used to print anything on screen, same as **printf** in C language.
- ❑ **cin** and **cout** are same as **scanf** and **printf**, only difference is that **you do not need to mention format specifiers like, %d for int etc, in cout & cin**.

## Comments in C++ Program

- 1) Single line Comment : //
- 2) Multi Line Comment : /\* \*/

## C++ Input and Output operator:=

- ❑ C++ I/O operation is using the stream concept. **Stream is the sequence of bytes** orflow of data. It **makes the performance fast**.
  - ❑ If bytes flow from **main memory to device like printer**, display screen, or a networkconnection, etc, this is called as **output operation** .
  - ❑ If bytes flow from device like printer, display **screen, or a network connection, etc to main memory**, this is called as **input operation**.
  - ❑ There are two stream function available in c++
- 1) Cout <<
  - 2) Cin >>

### Standard Output Stream (Cout) :

- ❑ The **cout** is a predefined object of **iostream** class.
- ❑ It is **connected with the standard output device**, which is usually a display screen.
- ❑ The cout is used in conjunction with stream **insertion operator (<<)** to display the output on a console

#### **Example:**

```
#include <iostream.h>
int main( )
{
    char ary[] = "Welcome to C++
    tutorial";cout << ary << endl;
}
```

#### **Output:**

Value of ary is: Welcome to C++ tutorial

### Standard input Stream (Cin) :

- ❑ The **cin** is a predefined object of **iostream** class. It is **connected with the standardinput device**, which is usually a **keyboard**.
- ❑ The cin is used in conjunction with stream **extraction operator (>>)** to read the input from a console.
- ❑ **Example:**

```
#include <iostream.h>
#include <conio.h>
void main( )
{
    int age;
    cout << "Enter your age: ";
    cin >> age;
```

```
    cout << "Your age is: " << age ;  
    Getch();  
}
```

Output:

Enter your age:

22Your age is: 22

## **Standard end line (endl)**

- ❑ The **endl** is a predefined object of **iostream** class. It is used to insert a new line characters and flushes the stream.

```
#include <iostream>  
#include <conio.h>  
void main( ) {  
    cout << "C++ Tutorial" << endl;  
    cout << " Javatpoint"<<endl;  
    cout << "End of line"<<endl;  
    getch();  
}
```

**Output:**

C++

Tutorial

Javatpoint

End of line

## **Structure Of C++ Program**

The structure of C++ program is divided into four different sections:

- (1) Header File Section
- (2) Class Declaration section
- (3) Member Function definition section
- (4) Main function section

### **Header File Section**

- ❑ This section contains various header files.
- ❑ You can include various header files in to your program using this section.  
For example:

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

- ❑ **Header file contains declaration and definition of various built in functions as well as object.**
- ❑ In order to use this built in functions or object we need to include particular headerfile in our program.

## Class Declaration Section

- ❓ This section contains declaration of class.
- ❓ You can declare class and then declare data members and member functions inside that class.

For example:

```
class Demo
{
    int a,
    b;
    public:
    void input();
    void output();
}
```

- ❓ You can also inherit one class from another existing class in this section.

keyword      user-defined name

```
class ClassName
{
    Access specifier:      //can be private,public or protected
    Data members;          // Variables to be used
    Member Functions() { } //Methods to access data members
};                          // Class name ends with a semicolon
```

## Object:=

### Declaring Objects:

- ❓ When a class is defined, only the specification for the object is defined; no memory or storage is allocated. To use the data and access functions defined in the class, you need to create objects.

### Syntax:

**ClassName ObjectName;**

### Accessing data members and member functions:

- ❓ The data members and member functions of class can be accessed using the dot('.') operator with the object. For example if the name of object is *obj* and you want to

access the member function with the name *printName()* then you will have to write *obj.printName()*.

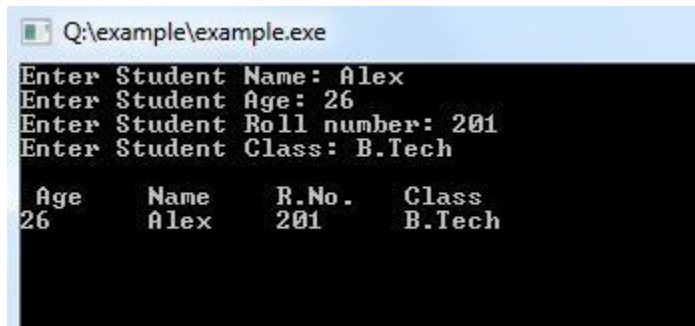
## Accessing the Data Members

- ❓ The public data members of objects of a class can be accessed using the direct member access operator (.).

# Program to Enter Students Details and Display it

Example:

Program Output:



```
Q:\example\example.exe
Enter Student Name: Alex
Enter Student Age: 26
Enter Student Roll number: 201
Enter Student Class: B.Tech

Age    Name    R.No.    Class
26     Alex    201     B.Tech
```

## C++ Program using Class:=

- ❑ **Classes name must start with capital letter**, and they contain data variables and member functions.
- ❑ This is a mere introduction to classes, we will discuss classes in detail throughout the C++ tutorial.

```
class Abc
{
    int i;                                //data variable
    void display()                        //Member Function
    {
        cout << "Inside Member Function";
    }
}; // Class ends here

int main()
{
    Abc obj; // Creatig Abc class's object
    obj.display(); //Calling member function using class object
}
```

### Different ways of defining member functions of a class

There are two ways in which the member functions can be defined :

- ❑ **Inside the class definition**
- ❑ **Outside the class definition**

## C++ class and functions: Inside the class definition

- ❑ As the name suggests, here the functions are defined inside the class.
- ❑ **Functions defined inside the class are treated as inline functions automatically** if the function definition doesn't contain looping statements or complex multiple line operations.



### Example

```
#include <iostream>
using namespace std;
class car
{
private:
    int car_number;
    char car_model[10];
public:
    void getdata()
    {
        cout<<"Enter car number: "; cin>>car_number;
        cout<<"\n Enter car model: "; cin>>car_model;
    }
    void showdata()
    {
        cout<<"Car number is "<<car_number;
        cout<<"\n Car model is "<<car_model;
    }
};
```

```
// main function starts
int main()
{
    car c1; c1.get-
    data();
    c1.showdata();
    return 0;
}
```

### Output

```
Enter car number : 9999
Enter car model : Sedan
Car number is 9999
Car model is Sedan
```

Here in above program `getdata()` and `showdata()` are the member function defined inside the class.

## C++ class and functions: Outside the class definition

- ❑ As the name suggests, here the functions are defined outside the class however they are declared inside the class.
- ❑ Functions should be declared inside the class to bound it to the class and indicate it as its member but they can be defined outside of the class.
- ❑ **To define a function outside of a class, scope resolution operator `::` is used.**

### ***Syntax for declaring function outside of class***

```
class class_name
{
    .....
    .....
    public:
        return_type function_name (args); //function declaration
};
//function definition outside class
return_type class_name :: function_name (args)
{
    .....; // function definition
}
```

### **Example**

```
#include <iostream>
using namespace std;
class car
{
    private:
        int car_number;
        char car_model[10];
    public:
        void getdata(); //function declaration
        void showdata();
};
// function definition
void car::getdata()
{
    cout<<"Enter car number: "; cin>>car_number;
    cout<<"\n Enter car model: "; cin>>car_model;
}
void car::showdata()
{
    cout<<"Car number is "<<car_number;
    cout<<"\n Car model is "<<car_model;
}
// main function starts
int main()
{
    car c1;
    c1.getdata();
    c1.showdata();
    return 0;
}
```

### **Output**

```
Enter car number : 9999
Enter car model : Sedan
Car number is 9999
Car model is Sedan
```

Here is this program, the functions `showdata()` and `getdata()` are declared inside the class and defined outside the class. This is achieved by using scope resolution operator `::`.

## Member Definition Section

- ❑ This section is **optional** in the structure of C++ program.
- ❑ Because you can define member functions inside the class or outside the class.
- ❑ If all the member functions are defined inside the class then there is no need of this section.
- ❑ This section is **used only when you want to define member function outside theclass.**
- ❑ This section **contains definition of the member functions that are declared inside theclass.**

### For example:

```
void Demo::input ()
{
    cout << "Enter Value of A:";
    cin >> a;
    cout << "Enter Value of B:";
    cin >> b;
}
Void Demo :: output()
{
    Cout << "The values of A is :"<<a;
    Cout << "The value of B is :"<<b;
}
```

## Main Function Section:

- ❑ In this section you **can create an object of the class and then using this object you can call various functions defined inside the class** as per your requirement.
- ❑ For example:  
Void main ()  
{  
 Demo d1;  
 d1.input ();  
 d1.output ();  
  
 Getch();  
}
- ❑ We can also compare the structure of C++ program with client server application. In client server application client send request to the server and server sends response to the client.

- ❑ In above C++ structure the class declaration section and member function definition section both together works as a server and **main () function section works as a client.**
- ❑ Because in main () function section we create an object of the class and then using that object we make a call to the function declared in the class.

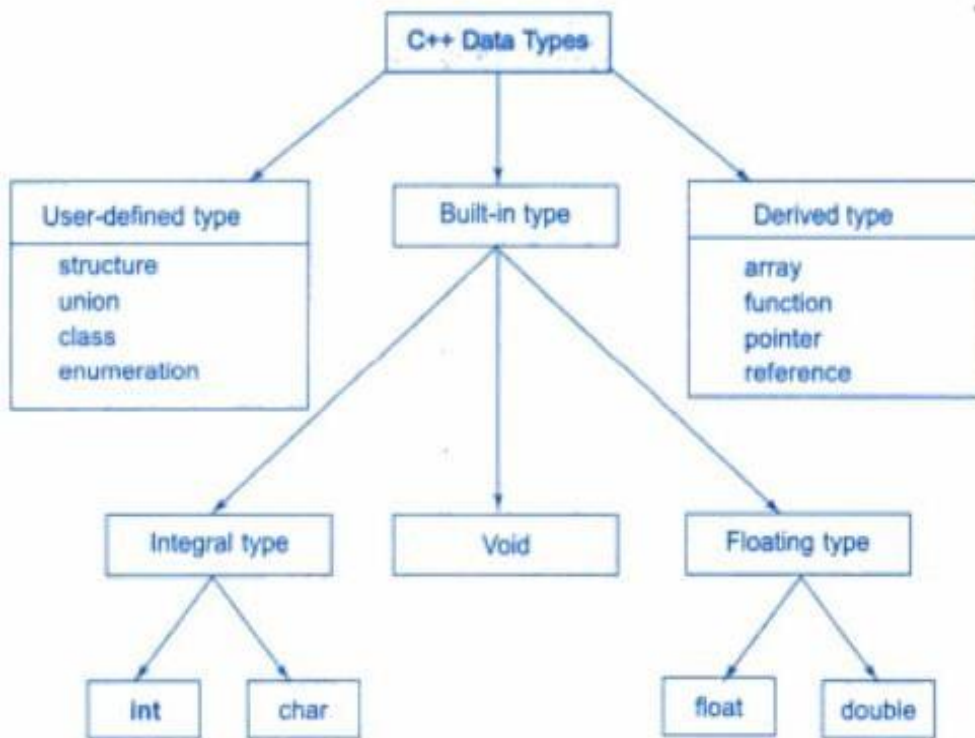
### **Example:**

```
#include
<iostream.h>
#include<conio.h>
Class test
{
    Int a,b;
    Public:
    Void input();
    Void display();
}
Void test :: input()
{
    Cout << "Enter value of A& B:";
    Cin >> a >> b;
}
Void test :: display ()
{
    Cout<< "The A:" <<a;
    Cout << "The B: << b;
}
Void main()
{
    Test t1;
    t1.input();
    t1.display();
    getch();
}
```

## **Tokens ,Expression and Control Structure**

**Tokens** are the smallest individual units in a program.

- ❑ Keywords
- ❑ Identifiers : name to program elements (cannot start with digit)
- ❑ Constants : fixed values that don't change during execution
- ❑ Strings Operators



### **3 uses of void:**

- 1) to specify return type,
- 2) to indicate empty argument list and
- 3) to declare generic pointers.

### **User Defined Data Types**

### **Enumerated Data Type**

- ❑ An enumeration is a user-defined data type that consists of integral constants. To define an enumeration, keyword **enum** is used.
- ❑ Here, the name of the enumeration is *season*.
- ❑ And, *spring*, *summer* and *winter* are values of type *season*.
- ❑ By default, *spring* is 0, *summer* is 1 and so on. You can change the default value of an enum element during declaration (if necessary).

### **Enumerated Type Declaration**

- ❑ When you create an enumerated type, only blueprint for the variable is created. Here's how you can create variables of enum type.

```
enum boolean { false, true };  
  
// inside function  
enum boolean check;
```

- ❑ Here, a variable *check* of type enum boolean is created.
- ❑ Here is another way to declare same *check* variable using different syntax.

```
enum boolean
{
    false, true
} check;
```

#### Example 1: Enumeration Type

```
#include <iostream>
enum week { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday };
void main()
{
    week today;
    today = Wednesday;
    cout << "Day " << today+1;
    getch();
```

```
}
```

#### Output

Day 4

#### Example2: Changing Default Value of Enums

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

enum seasons { spring = 34, summer = 4, autumn = 9, winter = 32};

int main() {

    seasons s;

    s = summer;
    cout << "Summer = " << s << endl;

    return 0;
}
```

#### Output

Summer = 4

## Derived Data Types

### Arrays:

- ❑ Difference between C and C++ strings >> in C++ the array dimension of the string should include the null character.

### Pointers

A pointer can be used to store the memory address of other variables, functions, or even other pointers. We can access and manipulate the data stored in that memory location using pointers.

Syntax : datatype \* ptr\_name;

### **Pointer Initialization :**

```
int var = 10;
```

```
int * ptr;
```

```
ptr = &var;
```

Example : var =10, &var=address of var, ptr=address of var, \*ptr=10

### Reference Variables

- ❑ Provides different names to the same variable. ALIAS (one location two names)

Data-type & reference-name = variable-name

Int & num = sum ;

**int &** means **reference to a float variable**

- ❑ It must always be initialized when it is declared
- ❑ Main application is passing arguments to functions.

## Default Arguments in C++ Functions

**Que: What do you mean by default argument? When it is useful?**

A default argument is a value provided in a function declaration that is automatically assigned by the compiler if the caller of the function doesn't provide a value for the argument with a default value.

### Example: Default arguments in C++

```
#include <iostream>
int sum(int a, int b=10, int c=20);
void main(){
    /* In this case a value is passed as
    * 1 and b and c values are taken from
    * default arguments.
    */
    cout<<sum(1)<<endl;
    /* In this case a value is passed as
    * 1 and b value as 2, value of c values is
```

```

        * taken from default arguments.
        */
    cout<<sum(1, 2)<<endl;
    /* In this case all the three values are
    * passed during function call, hence no
    * default arguments have been used.
    */
    cout<<sum(1, 2, 3)<<endl;
    return 0;
}
int sum(int a, int b, int c)
{
    int z;
    z = a+b+c;
    return z;
}

```

### **What is visibility modifier OR access specifiers? List out differentiate with proper example**

To provide the facility of security by means of data hiding, is get done by some access specifiers and these access specifiers are known as visibility modifiers or access specifiers.

These visibility modifiers are used to secure the data member and member function of a class.

There are three types of visibility modifiers 1. Public 2. Private 3. Protected

Public variables are variables that are visible to all classes.

Private variables are variables that are visible only to the class to which they belong.

Protected variables are variables that are visible only to the class to which they belong, and any subclasses. Deciding when to use private, protected or public variable is sometimes tricky. You need to think whether or not an external object (or program), actually needs direct access to the information. If you do want other objects to access internal data, but wish to control it, you would make it either private or protected, but provide functions which can manipulate the data in a controlled way.

### **What is encapsulation or what is data hiding? How it is implemented in C++?**

The wrapping up of data and function into a single unit (called class) is known as encapsulation. Data encapsulation is the most striking feature of a class. **The data is not accessible to the outside world, and only those functions which are wrapped in the class can access it.**

In general, encapsulation is the inclusion of one thing within another thing so that the included thing is not apparent. **Decapsulation is the removal** or the making apparent a thing previously encapsulated.

In OOP, encapsulation is the inclusion within a program object of all resources need for the object to function – basically, the methods and the data. The object is said to publish its interfaces. Other objects add here to these interfaces to use the object without having to be concerned with how the object accomplishes it.

An object can be thought of as a self –contained atom. The object interface consists of public methods and instantiated data.

Example :

```

#include <iostream.h>
#include <conio.h>
class Square
{
private:
int Num;

public:
void Get()
{

```



```

cout<<"Enter Number:";
cin>>Num;
}
void Display()
{
cout<<"Square Is:"<< Num * Num;
}
};

```

```

Void main()
{
Square Obj;
Obj.Get();
Obj.Display();
Getch();
}

```

### **Explain scope resolution operator with example.**

In C, the **global version of a variable can not be accessed from within the inner block**. C++ resolves this problem by introducing a new operator :: called the scope resolution operator. This can be used **uncover a hidden variable**.

Syntax ::variable name

Example:-

```

#include <iostream.h>
#include <conio.h>
int m= 10;//global m
main()
{
int m=20; //m redeclared, local to main
clrscr();
{
int k=m;
int m=30; //m declared again local to inner block
cout<<"we are inner block \n";
cout<<"k= "<< k << "\n";
cout<<"m= "<< m << "\n";
cout<<"::m=" <<::m << "\n";
}
cout<<"we are outer block \n";
cout<<"m= "<< m << "\n";
cout<<"::m=" <<::m << "\n";
}

```

Note:- It is to be noted ::m will always refer to the global m.

### **Explain Data Abstraction with example.**

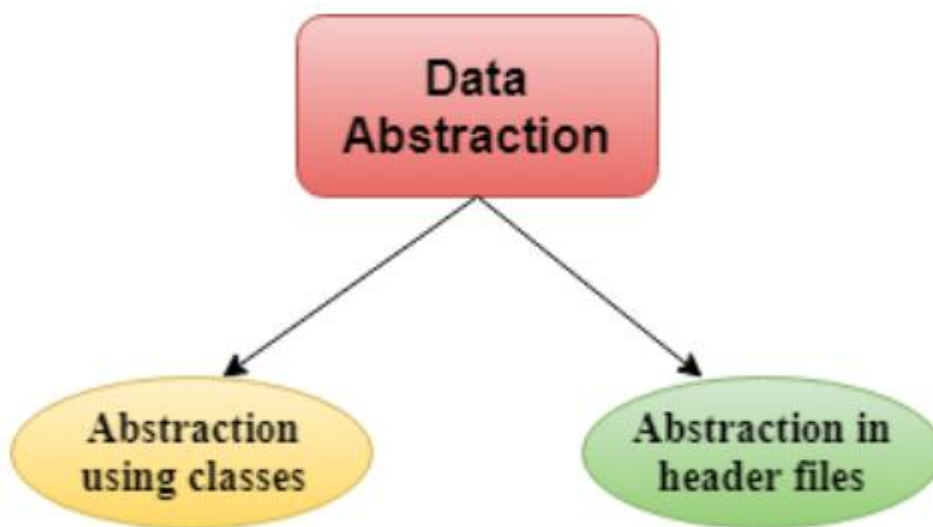
- Data Abstraction is a process of **providing only the essential details** to the outside world and **hiding the internal details**, i.e., representing only the essential details in the program.
- Data Abstraction is a programming technique that depends on the **separation of the interface and implementation details** of the program.

- Let's take a real life example of AC, which can be turned ON or OFF, change the temperature, change the mode, and other external components such as fan, swing. But, we don't know the internal details of the AC, i.e., how it works internally. Thus, we can say that AC separates the implementation details from the external interface.
- C++ provides a great level of abstraction. For example, `pow()` function is used to calculate the power of a number without knowing the algorithm the function follows.

In C++ program if we implement class with private and public members then it is an example of data abstraction.

**Data Abstraction can be achieved in two ways:**

- Abstraction using classes
- Abstraction in header files.



Abstraction using classes: An abstraction can be achieved using classes. **A class is used to group all the data members and member functions into a single unit by using the access specifiers. A class has the responsibility to determine which data member is to be visible outside and which is not.**

Abstraction in header files: Another type of abstraction is header file. For example, **`pow()` function available is used to calculate the power of a number without actually knowing which algorithm function uses to calculate the power.** Thus, we can say that header files hide all the implementation details from the user.

**Access Specifiers Implement Abstraction:**

- **Public specifier:** When the members are declared as public, members can be accessed anywhere from the program.
- **Private specifier:** When the members are declared as private, members can only be accessed only by the member functions of the class.

Let's see a simple example of abstraction in header files.

**// program to calculate the power of a number.**

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

```

int main()
{
    int n = 4;
    int power = 3;
    int result = pow(n,power);    // pow(n,power) is the power function
    cout << "Cube of n is : " << result << endl;
    return 0;
}

```

In the above example, pow() function is used to calculate 4 raised to the power 3. The pow() function is present in the math.h header file in which all the implementation details of the pow() function is hidden.

**Let's see a simple example of data abstraction using classes.**

```

#include <iostream.h>
class Sum
{
    private: int x, y, z; // private variables
    public:
    void add()
    {
        cout<<"Enter two numbers: ";
        cin>>x>>y;
        z= x+y;
        cout<<"Sum of two number is: " <<z<<endl;
    }
};
int main()
{
    Sum sm;
    sm.add();
    return 0;
}

```

In the above example, abstraction is achieved using classes. A class 'Sum' contains the private members x, y and z are only accessible by the member functions of the class.

## Advantages Of Abstraction:

- Implementation details of the class are protected from the inadvertent user level errors.
- A programmer does not need to write the low level code.
- Data Abstraction **avoids the code duplication**, i.e., programmer does not have to undergo the same tasks every time to perform the similar operation.

- The main aim of the data abstraction is to **reuse** the code and the proper partitioning of the code across the classes.
- Internal implementation can be changed without affecting the user level code.

**What is constructor? List out the characteristics of it. Explain with example.**

A constructor is a 'special' member function whose task is to initialize the objects of its class. It is special because its name is the same as the class name. constructor is involved is declared and defined as follow:

**Example:**

```
class integer
{
int m,n;
public:
integer()//declaring the constructor
{
m=25;n=96;
cout<<m<<endl<< n;
}
};
Main()
{
Clrscr();
Integer i1;
Getch();
}
```

**Characteristics:-**

They should be declared in the public section.

They are invoked automatically when the objects are created.

They do not have return types.

They can not be inherited, though a derived class can call the base constructor.

They have default arguments.

Constructor can not be virtual.

We can not refer to their address.

**What is parameterized constructor? How constructor called explicitly/implicitly?**

C++ permits us to achieve these objectives by passing arguments to the constructor function when the objects are created. The constructors that can take arguments are called parameterized constructor. We must pass the initial values as arguments to the constructor function when an object is declared. This can be done in two ways:

1. By calling the constructor explicitly.
2. By calling the constructor implicitly.

**Example :**

```
#include <iostream.h>
#include <conio.h>
class para_cons
{
int m,n;
public:
para_cons(int,int); //constructor declared
void display()
{
```

```

cout<<"m ="<< m << "\n";
cout<<"n ="<< n << "\n";
}
};
para_cons::para_cons(int x, int y) //constructor defined
{
    m=x; n=y;
}
main()
{
    clrscr();
    para_cons obj1(10,20); //constructor called implicitly
    para_cons obj2=para_cons(100,200); //constructor called explicitly
    cout << "\n OBJECT 1" << "\n";
    obj1.display();
    cout << "\n OBJECT 2" << "\n";
    obj2.display();

    getch();
}

```

### **Explain constructor with default argument?**

It is possible to define constructor with default arguments. In following example complex (float r, float i=0) The default value of the argument i is Zero. Then , the statement complex obj1(10.51) assigns the value 10.51 to r variable and 0.0 to i

### **Example :**

```

class complex
{
    float real,imag;
public:
    complex(float r,float i=0)//constructor declared
    {
        real=r;
        imag=i;
    }
    void display()
    {
        cout<<"Real=>"<< real << "\n";
        cout<<"Imag=>"<< imag << "\n";
    }
};
void main()
{
    clrscr();
    complex obj1(10.51); //constructor called implicitly
    cout << "\n OBJECT 1" << "\n";
    obj1.display();
    getch();
}

```

### **What is constructor? When copy constructor is called? What is an advantage of using copy constructor?**

A constructor is a 'special' member function whose task is to initialize the objects of its class. It is special because its name is the same as the class name.

Copy constructor is used to declare and initialize an object from another object. For example, the statement `B(A);`

Would define the object B and at the same time initialize it to the value of A.

Another form of statement is `B = A;`

The process of initializing through a copy constructor is known as copy initialization.

Remember the statement `B=A;` will not invoke the copy constructor. However, if B and A are objects, this statement is legal and simply assigns the values of A to B, member by member. This is the task of the overloaded assignment operator (`=`).

A copy constructor takes a reference to an object of the same class as itself as an argument. Let us consider a simple example of constructing and using a copy constructor as shown in program.

#### **Example :**

```
class copy_cons
{
    int id;
public:
    copy_cons(){}; //constructor declared
    copy_cons(int a) //constructor again
    {
        id=a;
    }
    copy_cons(copy_cons &x) //copy constructor
    {
        id=x.id;
    }
    void display()
    {
        cout<<id;
    }
};

void main()
{
    clrscr();
    copy_cons A(100); //Object A is created and initialize
    copy_cons B(A); //Copy constructor called
    copy_cons C=A; //Copy constructor called again
    copy_cons D; //D is created, not initialized
    D=A; //Copy constructor not called
    cout <<"\n id of A:=">A.display();
    cout <<"\n id of B:=">B.display();
    cout <<"\n id of C:=">C.display();
    cout <<"\n id of D:=">D.display();
    getch();
}
```

#### **Advantages**

A reference variable has been used as an argument of the copy constructor. We cannot pass the argument by value to a copy constructor.

When no copy constructor is defined, the compiler supplies its own copy constructor.

## Destructor

Destructor is an instance member function that is invoked automatically whenever an object is going to be destroyed. Meaning, a destructor is the last function that is going to be called before an object is destroyed.

- A destructor is also a special member function like a constructor. Destructor destroys the class objects created by the constructor.
- Destructor has the same name as their class name preceded by a tilde (~) symbol.
- It is not possible to define more than one destructor.
- The destructor is only one way to destroy the object created by the constructor. Hence destructor can-not be overloaded.
- Destructor neither requires any argument nor returns any value.
- It is automatically called when an object goes out of scope.
- Destructor release memory space occupied by the objects created by the constructor.
- In destructor, objects are destroyed in the reverse of an object creation.

The thing is to be noted here if the object is created by using new or the constructor uses new to allocate memory that resides in the heap memory or the free store, the destructor should use delete to free the memory.

### Example :

```
#include<iostream.h>
#include<conio.h>
int cCount = 0;
int dCount = 0;
class Test {
public:
    // User-Defined Constructor
    Test()
    {

        // Number of times constructor is called
        cCount++;
        cout << "No. of Object created: " << cCount
              << endl;
    }

    // User-Defined Destructor
    ~Test()
    {
        dCount++;
        cout << "No. of Object destroyed: " << dCount
              << endl;
        // Number of times destructor is called
    }
};
```

```
// driver code
int main()
{
    Test t, t1, t2, t3;

    return 0;
}
```

#### Output:

No. of Object created: 1  
 No. of Object created: 2  
 No. of Object created: 3  
 No. of Object created: 4  
 No. of Object destroyed: 1  
 No. of Object destroyed: 2  
 No. of Object destroyed: 3  
 No. of Object destroyed: 4

### Properties of Destructor

The following are the main properties of Destructor:

- The destructor function is automatically invoked when the objects are destroyed.
- It cannot be declared static or const.
- The destructor does not have arguments.
- It has no return type not even void.
- An object of a class with a Destructor cannot become a member of the union.
- A destructor should be declared in the public section of the class.
- The programmer cannot access the address of the destructor.

**Write major differences between constructor and destructor.**

Constructor	Destructor
1. Its name is same as class name.	1. Its name is same as class name preceded by ~ (tilde).
2. It is called automatically when an object is created.	2. It is called automatically when an object is destroyed
3. It can have any number of arguments.	3. It never takes any argument.
4. Normally new is used in constructor to allocate memory to objects.	4. Normally 'delete' is used in destructor to free memory from objects.

#### **What is Inheritance? Explain Types of Inheritance with example.**

☐The mechanism of deriving a new class from an old one is called inheritance (or derivation).The old class is referred to as the base class and the new one is called the derived class or subclass.

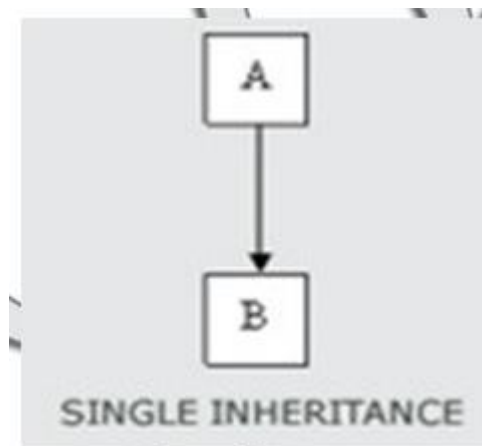
☐Inheritance is a mechanism of reusing and extending existing classes without modifying them, thus producing hierarchical relationships between them.



☑ Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and fast implementation time.

1. Single Inheritance
2. Multiple Inheritance
3. Hierarchical Inheritance
4. Multilevel Inheritance
5. Hybrid Inheritance

Single Inheritance: - Single Inheritance is method in which a derived class has only one base class.



Example

```
#include <iostream.h> class Value
{
protected:
int val;
public:
void set_values (int a)
{
    val=a;
}
};
class Cube: public Value
{
public:
int cube()
{
    return (val*val*val);
}
};
int main ()
{
    Cube cub;
    cub.set_values (5);
    cout << "The Cube of 5 is::" << cub.cube() << endl;
    return 0;
}
```

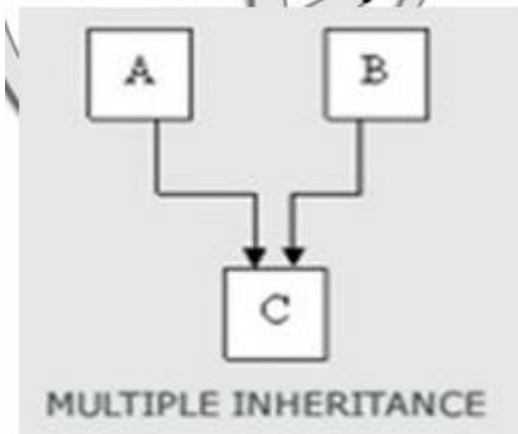
Result:

The Cube of 5 is:: 125

- ❑ In example shows a Base class value and a Derived class cube
- ❑ class value contain Protected member such As val
- ❑ Protected access Specifier we can use in Derived class.
- ❑ Private member can not be Inheriting.

**2. Multiple Inheritance:-** You can derive a class from any number of base classes. Deriving a class from more than one direct base class is called multiple inheritance.

- ❑ Multiple inheritance enables a derived class to inherit members from more than one parent. Deriving directly from more than one class is usually called multiple inheritance.



#### Syntax:-

```

Class C: Public A, Public B
{
  Body of D
}
  
```

#### Example

```

class student
{
  protected:
  int rno,m1,m2; public:
  void get()
  {
    cout<<"Enter the Roll no :"; cin>>rno;
    cout<<"Enter the two marks :"; cin>>m1>>m2;
  }
};

class sports
{
  protected:
  int sm; // sm = Sports mark public:
  void getsn()
  {
    cout<<"\nEnter the sports mark :"; cin>>sm;
  }
};

class statement:public student,public sports
{
  int tot,avg;
  public:
  
```

```

void display()
{
tot=(m1+m2+sm);
avg=tot/3;
cout<<"\n\n\tRoll No : "<<rno<<"\n\tTotal: "<<tot; cout<<"\n\tAverage   : "<<avg;
}
};
void main()
{
clrscr();
statement obj;
obj.get();
obj.getsm();
obj.display();
getch();
}

```

Output:

Enter the Roll no: 100 Enter two marks

90

80

Enter the Sports Mark: 90 Roll No: 100

Total : 260

Average: 86.66

Ambiguity: - In multiple inheritance the ambiguity arises when same method name is being used by two derived class and further derivation from these two base classes.

☑To resolve this ambiguity we are using scope resolution operator.

```

class M
{
public:
void display()
{
cout<<"vimal \n";
}
};
class N
{
public:
void display()
{
cout<<"Vaiwala\n";
}
}
class P:public M,public N
{
public:
void display(void)
{
M::display();

```

```

N::display();
}
};
int main()
{
clrscr();
P p;
p.display();
getch();
}

```

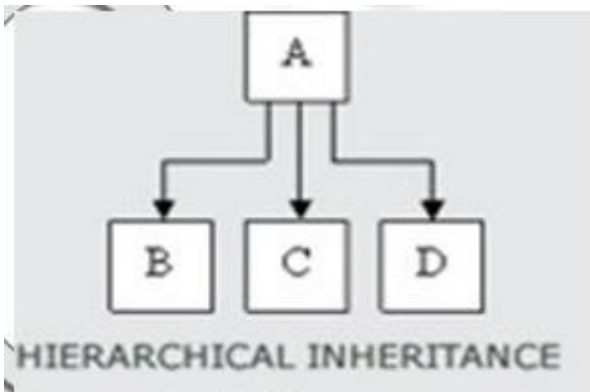
Output:  
Vimal Vaiwala

**3. Hierarchical Inheritance:** - It is the inheritance hierarchy wherein multiple subclasses inherit from one base class

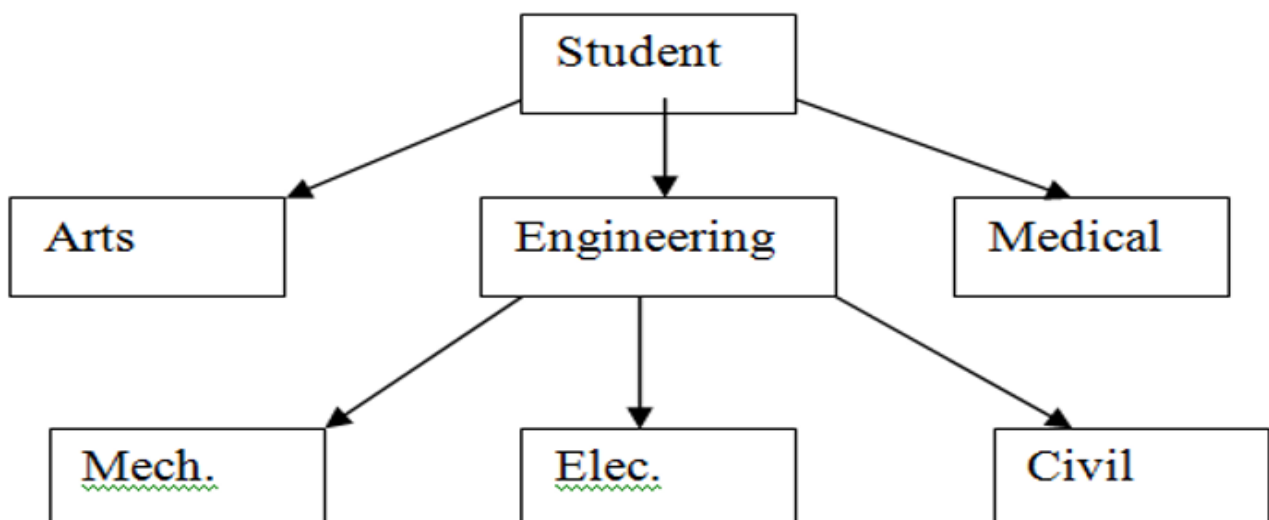
❑ Hierarchical Inheritance is a method of inheritance where one or more derived classes is derived from common base class.

❑ It is the process of deriving two or more classes from single base class. And in turn each of the derived classes can further be inherited in the same way.

❑ The base class will include all the features that are common to the subclasses. A subclass can be constructed by inheriting the properties of the base class. A subclass can serve as a base class for the lower level classes and so on.



Example of Hierarchical representation of student life in University



**Syntax:-**

```
class A
{

};
class B : visibility_label A
{

};
class C : visibility_label A
{

};
class D : visibility_label A
{

};
```

Here the visibility\_label can be private, protected or public. If we do not specify any visibility\_label then by default is private.

```
class A
{
public:
int val;
void getvalue(int j)
{
val=j;
}
};
class B : public A
{
public:
void square()
{
cout<<endl<<val*val;
}
};
class C : public A
{
public: void cube()
{
cout<<endl<<val*val*val;
}
void main()
{
int i;
clrscr();
B b;
```

```

C c;
cin>>i;
b.getvalue(i);
b.square();
c.getvalue(i);
c.cube();
getch();
}

```

Output

2

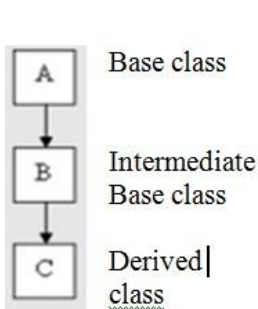
4

8

**4. Multilevel Inheritance:** - The mechanism of deriving a class from another 'derived class' is known as multilevel inheritance.

☐ It is the inheritance hierarchy wherein subclass acts as a base class for other classes.

☐ Multilevel Inheritance is a method where a derived class is derived from another derived class.



**Syntax:**  
class A{.....};  
class B: public A{.....};  
class C: public B{.....};

The class A serve as a base class for the derived class B, which in turn serves as a base class for the derived class C. The class B is known as intermediate base class since it provides a link for the inheritance between A and C. The chain ABC is known as inheritance path.

Example class A

```

{
protected:
    int x;
public:

void showA()
{
cout<<"enter a value for x:"<<endl;
cin>>x;
}
};
class B:public A
{
protected:
    int y;
public:
void showB()
{
cout<<"enter value for y:";
cin>>y;
}
}

```

```

}
};
class C:public B
{
public:
void showC()
{
showA();
showB();
cout<<"x*y ="<<x*y;
}
};
void main()
{
C ob1;
ob1.showc();
}

```

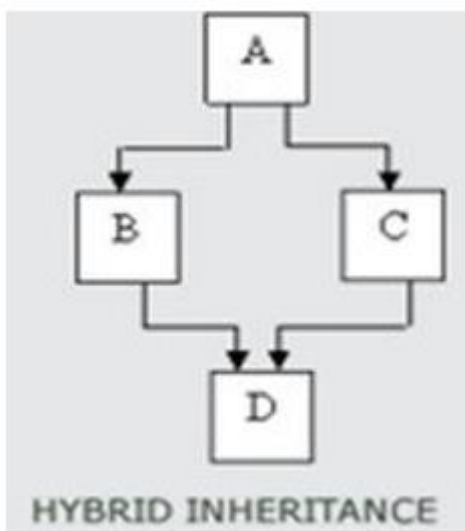
Output

Enter value of x = 12 Enter value of y = 3 X \* Y =36

**5. Hybrid Inheritance:** The inheritance hierarchy that reflects any legal combination of other four types of inheritance.

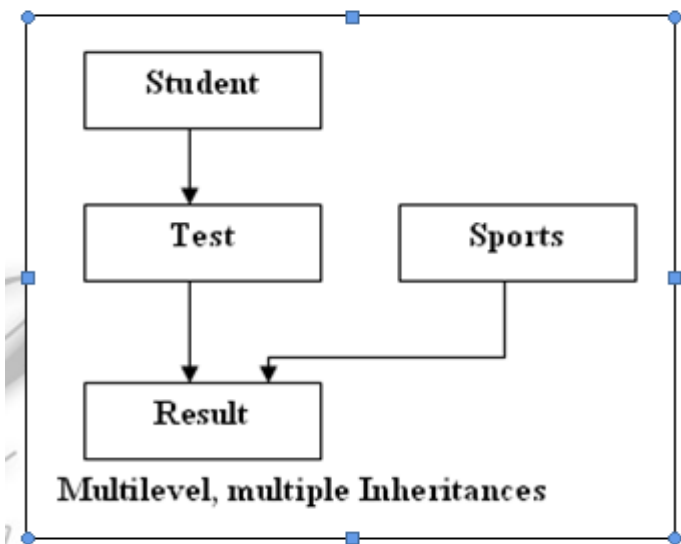
There could be situations where we need to apply two or more types of inheritance to design a program.

"Hybrid Inheritance" is a method where one or more types of inheritance are combined together and used



In figure base class A and class B and C are derived from class A. So that path is called Hierarchical Inheritance.

Class B and C are the base class for class D. so that path is called multiple inheritance. because there are more then one base class.



Example:- class stu

```

{
protected:
int rno;

public:

{
rno=a;
void get_no(int a);
}

void put_no(void)
{
out<<"Roll no"<<rno<<"\n";
}
};
class test:public stu
{

public:

protected:
float part1,part2;

void get_mark(float x,float y)
{
part1=x;
part2=y;
}
void put_marks()
{
cout<<"Marks obtained:"<<"part1="<<part1<<"\n"<<"part2="<<part2<<"\n";
}
};
class sports
{
public:

```



```

protected:
    float score;

void getscore(float s)
{
    score=s;
}
void putscore(void)
{
    cout<<"sports:"<<score<<"\n";
}
};

class result: public test, public sports
{

public:

};
float total;

void display(void);

void result::display(void)
{
    total=part1+part2+score;
    put_no();
    put_marks();
    putscore();
    cout<<"Total Score="<<total<<"\n";
}
int main()
{
    clrscr();
    result stu;
    stu.get_no(123);
    stu.get_mark(27.5,33.0);
    stu.getscore(6.0);
    stu.display();
    return 0;
}

```

#### OUTPUT

Roll no 123

Marks obtained : part1=27.5 Part2=33

Sports=6

Total score = 66.5