Dr Subhash Technical Campus Junagadh

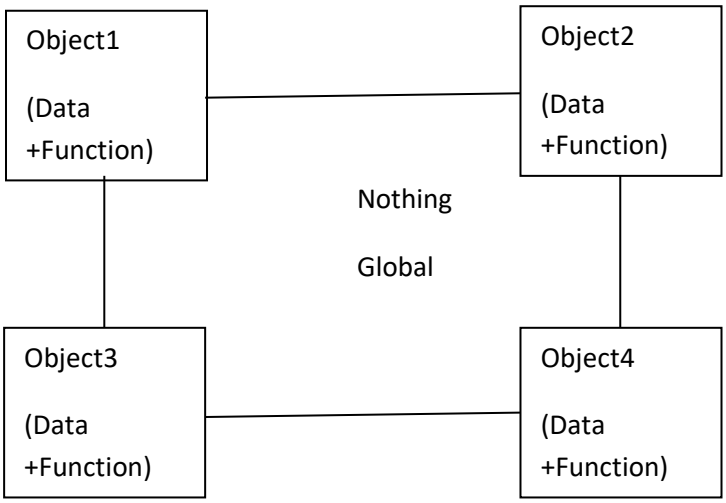
Programming In C++

Diploma In Computer Engineering

Prepared By Chirag Kaneria

What is OOP. Give advantages and applications of OOP.

- OOP means Object Oriented Programming.
- In OOP, program is divided into small parts called objects. So objects is basic building block of Object Oriented Programming.
- Objects giving primary importance to data defining object structures in OOP.
- The functions of an objects are called the member function.



- As you can seen in following figure four objects object1, object2, object3, object4 are connected with each other using their member function.
- The objects can communicate with each other by calling one another's member functions .
- Functions that operates on the data of an objects are placed in same unit.
- So data is not freely accessible to other objects means data is hidden and hence misuse is protected.
- New data and functions can be easily added whenever necessary.
- Some Object oriented languages are C++,JAVA,C#

Advantages of Object Oriented Programming.

- OOP offer several advantage to the developers in term of making software cost effective, easy to maintain, high quality and greater programmer productivity.
- User defined data types can be easily constructed.
- OOP provides data hiding so provides more data security

- Existing code can be reused in other application by means of inheritance. So in this way inheritance save time and memory.
- Objects-oriented systems can be easily upgraded from small to large systems.
- It is way to partition the work in a project on objects.
- Using OOP concept designing and software complexity can be easily managed.
- With OOP software complexity can be easily managed.

Application of Object Oriented Programming.

Some of application area of OOP are in:

- Personal software
- Object-oriented Database systems
- AI ,expert systems
- CIM/CAM/CAD systems
- Enterprise systems
- Simulation and modelling
- Web technologies
- Neural networks and parallel programming.

Difference between OOP and POP

	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 procedures 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.
Examples	Example of POP are : C, VB, FORTRAN, Pascal.	Example of OOP are : C++, JAVA, VB.NET, C#.NET.

Explain basic concept of Object Oriented Programming.

There are following basic concept of Object Oriented Programming:

1) Objects
2) Classes
3) Abstraction
4) Encapsulation and data hiding
5) Inheritance
6) Polymorphism
7) Message passing

Objects: Objects are basic run time entities in object-oriented systems.

Objects are variables of type class. And Object are instance of class

Class: Objects with same properties and operations create a group known as class.

Class is a user defined data type in C++

For example mango, apple and orange are members of the class fruit

Abstraction: It refer to the act of representing necessary features without include explanation. This property is implemented by using the classes in C++ .

Encapsulation and data hiding: Putting data and function into a single unit is known as Encapsulation.

Thus, the data of the object is hidden from the rest of world and known as **Data Hiding**

Inheritance: Inheritance is a process of one class inherit the properties of the another class.

Class which gives property to another class is called **base class or parent class or super class.**

Class which takes property to another class is called **derived class** or **child class** or **sub class**

There are five type of inheritance:

I. Single level Inheritance
II. Multilevel Inheritance
III. Multiple Inheritance
IV. Hierarchical Inheritance
V. Hybrid Inheritance

Polymorphism:

- Polymorphism is also called overloading.
- The ability to act differently in different situation.
- Function or an operator can act differently in different situation is called polymorphism
- There are two type of polymorphism:
  - **Compile time Polymorphism (Early binding or Static binding)**
    - In Compile time Polymorphism function call bind its suited function definition at compile time
    - Operator overloading and function overloading are examples of compile time polymorphism
  - **Run time Polymorphism (Dynamic binding or Late binding)**
    - In Run time Polymorphism function call bind its suited function definition at run time
    - We can implement Run time Polymorphism using virtual function.
- + operator can work differently in with two integer number and two string.

Ex. **8+2=10** //Addition
"jay"+"patel"="jaypatel" //String concatenation

Explain Input/output operator.

Output/insertion operator (<<)

- The keyword '**cout**' is a pre-defined object that represents the standard output stream in C++.
- The operator << is called **insertion operator or putto** which is used to print data on to the monitor

Example :

```
Cout<<"hello";

Cout<<"a:" <<a;
```

Input/Extraction operator (>>)

- The keyword '**cin**' is a pre-defined object that represents the standard input stream in C++.
- The operator >> is called **extraction operator or get from operator** which is used extract the value from user.

Example :

```
Cin>>a;

Cin>>a >>b
```

Explain reference variables with example

- Reference variable are not supported in C and supported only in C++.
- They are used create alternative names for previously defined variable.
- The reference refers to the same memory location and hence they represent the same value by more than one name.
- The reference variable must be initialized at the time of declaration.
- It is important to note that the initialization of a reference variable is completely different from assignment to it.
- The syntax for reference variable is as follow
  Data_type& reference_name= variable_name;
- Example:

```
int x=10;
```

```
int&  y=x;
```

- Here x is original variable having value 10. The y is an alternative name of x and hence the value of y is also 10.
- So if you update the x as

```
x=x+1;
cout<<x<<y;
```

we will get 11 as **x** (as well **y**) is incremented.
The same is true for opposite way means updating **y** also updates **x**.

Program of reference variable:

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

void main()
{
int a=10;
int& b=a;
a++;
cout<<"value of a:"<<a;
cout<<"value of b:"<<b;

getch();
}
```

Output:

```
value of a:11
value of b:11
```

Explain scope resolution operator with example

- According to scope there are two type of variable
  1)local variable 2)global variable
- Local variables defined in function are only accessible within that function where global variables are accessible in whole program
- It is possible that if two variables are defined with same names in different scope area, there scope overlaps and only local most is accessible in overlapping area.

- For example:

```
int x=10;      //global variable
void main()
{
int x=20;      //local variable
cout<<x;       // local is accessed
}
```

- In above example, when x is printed by cout, it prints 20 as value of local x is 20. But how can we access value of global x.
- When local variable and global variable have same name, then C++ provides **scope resolution operator(::)** to access global variables
- To access global variable, variable name is preceded by **::** as

  :: variable_name
- We can use scope resolution operator for defending member function outside class

Program

```
#include<iostream.h>
#include<conio.h>
int x=10;      //global variable
void main()
{
int x=20;      //local variable
cout<<"local variable value: "<<x;      // local is accessed
cout<<"global variable value:"<<::x   //global is accessed
}
```

Output:

```
local variable value:20
global variable value:10
```

Explain memory management operator with example

- C use library functions malloc() and free() to allocate and deallocate the memory dynamically at run time.
- C++ provides the in-built operators **new** and **delete** for dynamic memory management.
- The new operator is used to allocate memory and delete operator is used to deallocate the memory
- The syntax for allocating memory using **new** as follows

  Pointer_variable = new data_type;

Example:

```
int *p;

p = new int;
```

in example allocates the memory to store one integer

- To give initial value while allocating memory, following format is used.

  Pointer_variable = new data_type(initial_value);

Example:

```
int *p = new int(12);
```

- The syntax for allocating memory for array using **new** as follows

  Pointer_variable = new data_type[array_size];

Example:

```
int *p = new int[5];
```

given example allocates memory for array of 5 integer

- We can deallocate memory or remove the variable created dynamically using **delete** Operator.
- Remember that delete operator removes the memory pointed by pointer , but pointer itself is not removed.
- To deallocate memory for single data, following syntax is used.

  delete pointer_variable;

Example: **delete p;** //removes the variable pointed by pointer p.

- To deallocate memory for array, following syntax is used.

  delete[ ] pointer_variable;

  Example: **delete [ ] p;** // removes the array pointed by pointer p.

```
#include <iostream.h>
#include <conio.h>
void main( )
{
        int *p;
        p=new int;
        cout<<"p="<<p;
        *p=25;
        cout<<"value="<<*p;
                delete p;
                  getch();
                }
```

Output:

p=0x8dhdcdc2

value=25

Difference between call by value, call by address, call by reference:

call by value	call by address	call by reference
Value of actual parameter are passed to formal parameter	Address of actual parameter are passed to formal parameter	References are created for actual parameter.
Need extra memory to store copy of value	Need extra memory to store copy of address	No Need extra memory as references are created
Change in formal parameter does not change in actual parameter	Change in formal parameter can change in actual parameter	Change in formal parameter can change in actual parameter

Slower as values are to be copied	Slower as addresses are to be copied	Faster as no memory operation required.
Simple and easy to understand	Complex as pointers are involved	Simple and easy to understand
Supported by C and C++ both	Supported by C and C++ both	Supported only by C++

Explain call by reference

- call by reference is a parameters passing techniques in C++.
- call by reference parameter passing is not supported in C.
- when function is called , the caller creates a reference for each argument and using reference the actual parameters area accessed , this method of passing the parameters to the function is called call by reference.
- so we can say that reference are created for actual parameter
- formal parameter can change the actual parameter.
- actual parameter is read-write and actual parameter must be a variable in call by reference
- As compare to call by value and call by address, call by reference is faster because no memory operation required
- As compare to call by address, call by reference is simple and easy to understand.

Example of call by reference:

```
#include<iostream.h>

#include<conio.h>


void change(int& a, int& b);                    //prototype, & indicate address

void main()

{

int x=10,y=20;

cout<<"Before function call  values are:"<<"x="<<x<<"y="<<y;
```

```
change(x,y);
cout<<"After function call  values are:"<<"x="<<x<<"y="<<y;
}
void change(int& a, int& b)
{
a++;
b++;
cout<<"In function   values are:"<<"a="<<a<<"b="<<b;
}
```

output:

Before function call values are: x=10 y=20

In function call values are: a=11 b=21

After function call values are: x=11 y=21

Return by reference

- A function can also return a reference, now take a following function

```
#include<iostream.h>
#include<conio.h>
int& max(int& a, int& b);              //prototype, & indicate address
void main()
{
int x=10,y=20,m;
m=max(x,y);
cout<<"maximum number is"<<m;
getch();
```

```
}
int& max(int& a, int& b)
{
if(a>b)
   return a;
else
   return b;
}
```

here return type of max() is int&, the function return reference to x or y(not value) then function call such as max(x,y) will reference to either x or y depending on their values.

Explain inline function with example

- Function save memory space because all calls to function cause the same code to be executed, the function body need not be duplicated in memory.
- When a compiler see a function call, it normally generate a jump to the function definition, at end of the function, it jump back to the instruction after call.
- But when every time a function is called, it takes a lot of extra time in executing a series of instruction for task such as jumping to the instruction.
- One solution for speedy execution and saving memory space is inline function in C++. It is used only for short function.
- When we use inline function, function call replaced by inline function body, instead of the control going to the function definition and coming back.
- You can declare an inline function before main() by just placing the keyword **inline** before a normal function

- Syntax for declaring inline function:

```
inline  return_type  function_name(argument list)
{
        Function body
}
```

Advantage:

- Reduces the execution time and save memory.
- An inline function is about 10000+ more faster than normal function.
- Improves program readability and modularity

Disadvantage:

- inline function must be defined before use.
- Increase size of the executable file, it also effects file size
- The following situation where inline function are not used:
    1. If function contain static variable.
    2. It is not used any loop, switch, goto statement
    3. Recursive function

Program:

```
#include<iostream.h>

#include<conio.h>

inline int sum(int x, int y)

{

        int c=a+b;

        return c;

}

void main()

{

        int a=3,b=12;

        cout<<sum(a,b)
```

```
    getch();
}
```

Function overloading with example.

- Function or an operator can act differently in different situation is called polymorphism
- Function overloading means more than one functions with same name ,different return type and different number of arguments
- C does not allow two function with same name. While in C++ we can have more than one function with same name.
- In Compile time Polymorphism function call bind its suited function definition at compile time
- This called early binding or static binding or compiles time polymorphism.
- Function overloading and operator loading are examples of compile time polymorphism.
- For example an overloaded add() function handles different types of data as shown below:

```
// declarations
int add(int a, int b);                    //prototype-1
int add(int a, int b, float c);           //prototype-2
 float add(float  a, int b);              //prototype-3
float add(int   a, float b);              //prototype-4
```

Program of use of function overloading:

```
#include<iostream.h>
#include<conio.h>
int area(int a)
{
        return(a*a);
}
int area(int a,int b)
{
        return(a*b);
}
```

```
void main()
{
        int  l,b;
        cout<<"enter the length:"
        cin>>l;
        cout<<"area of square="<<area(r)<<endl;
        cout<<"enter the length and breadth:";
        cin>>l>>b;
        cout<<"area of rectangle="<<area(l,b);
        getch();
}
```

output:

enter the length 8

area of circle=64

enter the length and breadth:4

5

area of rectangle=20

Explain Default argument in function

- Default value are specified when the function is declared ,the compilers look at the prototype to see how many arguments as a function
- Default value from right to left only ,Not in the middle of an argument
- For example int max(int a, int b=10, int c=20) \\legal statement
  int max(int a, int b=10, int c) \\illegal statement
  int max(int a=20, int b=10, int c) \\illegal statement
  int max(int a=20, int b=10, int c=30) \\legal statement

Program:

```
#include<iostream.h>
#include<conio.h>
void sum(int x, int y=20,int z=30);

void main( )
{
```

```
    clrscr();
    sum(100);
    sum(100,50);
    sum(100,80,50);


    getch( );
}
void sum(int x, int y, int z)
{
    int ans=x+y+z;
    cout<<ans<<endl;

 }
```

OUTPUT: 150

180

230

Explain class with example

- Class is a user defined data type.
- Objects with same properties and operations create a group known as class.
- example mango, apple and orange are members of the class fruit
- the variable declared inside the class are known as data member , and the functions are known as member function.
- Syntax:

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

- Example

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

class Rectangle
{
   int width, height;
  public:
   void set_values (int l,int b)
        {
        width=l;
        height=b;
        }
   int area()
        {
        return width*height;
        }
};


void main () {
  Rectangle rect;
  rect.set_values (3,4);
  cout << "area: " << rect.area();
 getch();
}
```

Explain access modifier with example

- Access modifier is supported in C++ and not in C.
- Access modifier is also known as visibility label and access specifiers .
- All the members of class fall under one of the following three access modifier (visibility label):
  1. **public**
  2. **private**
  3. **protected**

- public data member and member function of class can be accessed in whole program
- private data member and member function of class are only within class and friend function.
- protected data member and member function of class are only within class, friend function and derived class of that class.
- Remember that by default is private in class.

Difference between class and structure

Class	Structure
All the member are private by default in class.	All the member are public by default in structure
Class provides data hiding by using private, public, and protected specifiers for members	Structure does not provide data hiding
With classes we can implement the concept of inheritance	With structure we can not implement the concept of inheritance

Explain static data member and static member function

Static data member:

- C++ supports static data member or also called **class variable** is a member of all objects of class whose value same .

The characteristic of static data member are as follows

- It is allocated memory only once and shared among all the objects of a class.
- It is declared inside the class. But defined and initialized outside the class using scope resolution operator.
- It is allocated memory and initialized, when class is loaded. That means it exits even before any object of that class is created.
- The following class declaration show the syntax for declaring and defining the static data members.

```
class crow
{
   private:
        char location[20];
        float age;
        static char color[10];     //declaration

   public:

        ----------

        ----------

};

char crow::color[10]=”black”;
```

static member function:

- C++ supports the static function members. The characteristic of a static member function.
- it is defined with word **static**.
- Only one copy of static member function is created for a class.
- It is called with class name followed by scope resolution operator as
  class_name::static_function(arguments);
- It is restricted to use only static data member as it is called for class not for instance.

Program of use of static data member and static member function

```
#include<iostream.h>
#include<conio.h>
class student
{
        private:
                static int count;                       //static data member
```

```
public:
        student( )
        {
                count++;
        }
        static void get_count()              //static member function
        {
                cout<<count;
        }


};

int student :: count = 0;

void main()

{

cout<<" number of object created ";

student:: get_count();

student s1,s2,s3;

cout<<" number of object created";

student:: get_count();

getch();

}
```

Output:

number of students:0

number of students:3

Explain friend function with example

- C++ provides a special member function called friend function which is not member of class, but it is friend to a class.
- Private member of a class are accessible only inside a class, friend function which are not member of class, but it can use private members of class.
- To make a friend to a class, only function prototype is written in class with keyword **friend**, while friend function is defined outside the class. The syntax for declaring friend function are:

  friend return type function_name(argument list);
- Generally , friend function has the objects as arguments.
- unlike member functions, it can not access the member names directly and has to use an object name and dot operator with each member name(e.g A.x)
- A function can be declared as friend for any number of classes. It can not be member function of any class. It have full rights to access private data member of the class
- Generally it can be declared either public or private section of the class without effecting its meaning.

  .
- friend function is called like a normal function without the help of any object
- We can access the members without inheriting the class.
- Using friend function in C++ is a big disadvantage. Friend function is only function which are not member of any class and can access private variables of class, this breaks the concept of **data hiding** in OOP
- That is reason so that java is not support the friend function.

Example of use of friend function:

```
#include<iostream.h>
#include<conio.h>
class one
{
        int x;
        public:
                void get()
                {
                        cout<<"Enter value of private member x="
```

```
            cin>>x;
        }
        friend void display(one);
};
void display(one p)
{   cout<<" value of private member x="<<p.x;
}
void main()
{   one a;
    a.get();
    display(a)
    getch();
    }
```

Output:

Enter value of private member x=25

value of private member x=25

Constructor:

- Constructor is a special member function which is responsible for allocating memory to objects and used to initialization at the time of creation.

  Characteristics of constructor:
- Constructor is called automatically when object is created.
- Constructor's name is same as the class name.
- Constructor can not have return type.
- Constructor can be virtual.
- Constructor should be declared in the public section of class.
- they make implicit call to the new operator when memory is allocated.

they are following four types of constructor:

1. **default constructor**
2. **parameterized constructor**
3. **copy constructor**
4. **dynamic constructor**

Default constructor:

- default constructor has no argument.
- default constructor's body is null
- default constructor is called automatically when object is created(without values)
- If default constructor is not written then C++ compiler automatically supplies the default constructor.
- syntax of default constructor:

```
class_name()
{

}
```

Parameterized constructor:

- The C++ supports constructor with the arguments, known as parameterized constructor.
- The advantage of using parameterized constructor is that we can pass the values at the time of creating the objects with which the compiler automatically calls it.

```
#include <iostream>
#include<conio.h>
class point
{
  int x,y;
public:
```

```
        point()
        {
                x=0;
                y=0;
        }
        point(int i, int j)
        {
                x=i;
                y=j;
         }

 void show()
{
   cout << "x="<<x << "y="<<y<<endl;
 }
};
void  main()
{       point p1,p2(3,6);
        p1.show();
        p2.show();
        getch();
}
```

Output:

x=0 y=0

x=3 y=6

Copy constructor:

- copy constructor is used to create the new object with values of previously created object.

```
#include <iostream>
#include<conio.h>
class point
{
        int x, y;
public:
point( )
{
        x=0;
        y=0;
}
 point(int i, int j)
{
         x=i;
        y=j;
}
  void show()
{
    cout << x << " " << y;
  }
point(point& p)
        {       x=p.x;
                y=p.y;
        }
};
void  main()
{
  point p1(3,4);
point p2(p1);
```

```
point p3=p2;

p1.show();
p2.show();
p3.show();
  getch();
}
```

- the following statement creates object **p2** with values of **p1**

```
point p2(p1);                // call copy constructor
```

in other way

```
point p2=p1                  //call copy constructor


point(point& p)
{
        x=p.x;
        y=p.y;
}
```

- **point p2(p1)** calls copy constructor automatically and passes the reference of object p1.
- the statement in body of copy constructor copies the data member of p1 to the data members of p2

Dynamic Constructor:

- data member of a class is pointer which is used to point to the memory allocated to object dynamically using new operator.
- Dynamic constructor is required normally when object of a class are not of same size, but their size is determined at run time.
- **Example :**

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

```
class array
{
        int *a;
        int size;
public:
        array( )
        {
        size=0;
        a=new int[size];
        }
        array(int n )
        {
        size=n;
        a=new int[size];
        }
        void get( )
        {
                for(int i=0;i<size;i++)
                {
                        cin>>a[i];
                }
        }
        void show( )
        {
                for(int i=0;i<size;i++)
                {
                        cout<<a[i]<<endl;
                }
        };
void main( )
{
        int  n,i;
        cout<<"Enter Size of array";
        cin>>n;
```

```
    array a=new array(n);
    array a=new array(n);
    a.get( );
    a.show( );
    getch( );
}
```

constructor overloading

- Function overloading means more than one functions with same name for different return type and different number of argument.
- So Similar to function overloading we can create more than one constructor with different number of argument in a single class. This type of overloading is called constructor overloading.
- For Constructor ,its name is same as the class name. Constructor can not have return type and should be declared in the public section of class
- **Example:**

```
#include <iostream>
#include<conio.h>
class point
{
  int x,y;
public:

        point()
        {
                x=0;
                y=0;
        }
        point(int i, int j)
        {
                x=i;
```

```
                y=j;
        }

 void show()
{
   cout << "x="<<x << "y="<<y<<endl;
 }
};
void  main()
{       point p1,p2(3,6);
        p1.show();
        p2.show();
        getch();
}
```

Output:

x=0 y=0

x=3 y=6

Destructor:

- The destructor is a special member function which used to destroy the object which no longer needed then.
- The destructor's name is same as the class name but is preceded by a**'~'(tiled)** sign
- The destructor is automatically is called when object is destroyed .
- destructor does not have return type and argument.
- The destructor is called to release the memory space occupied by the object.

program:

`

```
#include<iostream.h>

#include<conio.h>
```

```
class abc
{
public:

abc( )
{
cout<<"constructor called";
}
~abc()
{
cout<<"destructor called";
}
};
void main()
{
        {
        abc  x;

                {
                        abc  y;
                }
        }
getch();
}
```

output:

```
constructor called
constructor called
destructor called
destructor called
```

Difference between constructor and destructor:

Constructor	Destructor
Constructor is used to initialize the objects of a class	The destructor is used to destroy the object which no longer needed then.

Constructor is called automatically when object is created	destructor is called automatically when object is deleted.
Constructor allocates the memory	destructor releases the memory
Constructor can have arguments	destructor can not have arguments
Overloading of constructor is possible	Overloading of destructor is not possible
Constructor's name is same as the class name	The destructor's name is same as the class name but is preceded by a**'~'(tiled)** sign

Classname(argument)
{
//body of constructor
}
	~Classname()
	{
	
	}

INHERITANCE:

Definition: Inheritance is process by which object of one class gets the properties of another class.

Inheritance means one class inherit the properties of the another class.

Base class(super class): The class from which another class gets properties is called base class.

Derived class(sub class): The class who getting properties from base class is called derived class.

The general form of inheritance is :

class derived_classname: **public/private/protected** base_classname

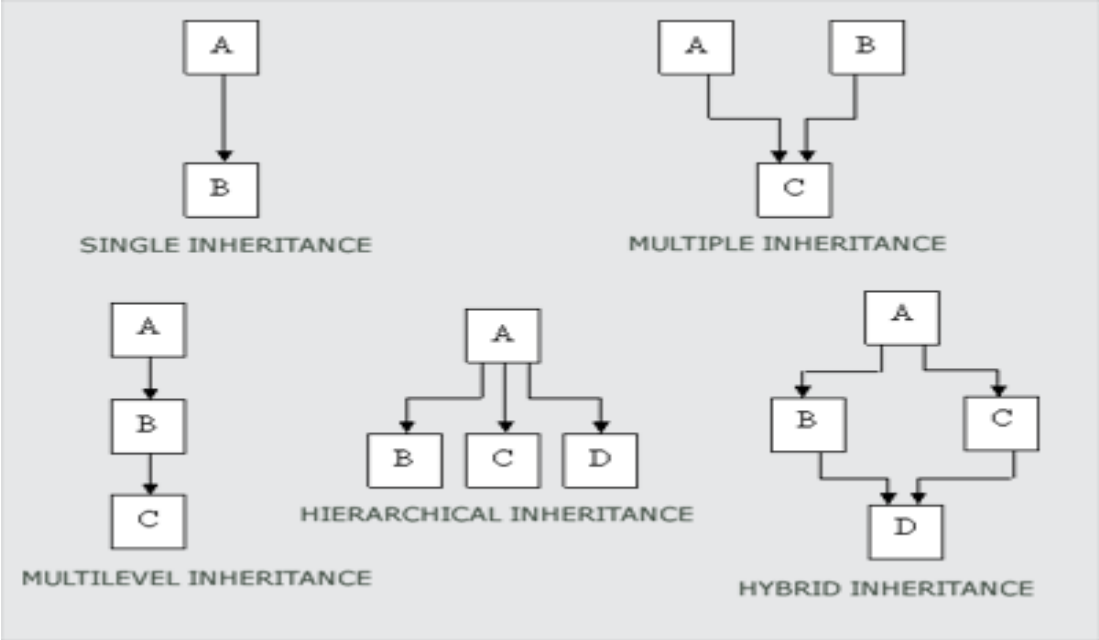
{.....};

Advantages of inheritance:

1. Inheritance are used to unable polymorphism.
2. Inheritance provide code reusability.
3. Elimination of redundant code.
4. Inheritance increase reliability of program.
5. Saves time and cost of program by reusability

Types of Inheritance

There are **five types** of inheritance:



I. **Single level Inheritance**
In this type of inheritance one derived class inherits from only one base class. It is the most simplest form of Inheritance.

II. **Multiple Inheritance**
In this type of inheritance, one derived class inherits from more than one base class

III. **Multilevel Inheritance**
In this type of inheritance, derived class inherits from base class and the derived class act as base class for another class

IV. **Hierarchical Inheritance**
In this type of inheritance more than one derived class inherits from only one base class.

V. **Hybrid Inheritance**
Hybrid Inheritance is a combination of Hierarchical and Mutilevel inheritance

Type of Inheritance as per access modifier:

When deriving a class from a base class, the base class may be inherited through public, protected or private inheritance. The type of inheritance is specified by the access-specifier as explained above.

Public Inheritance: When deriving a class from a public base class, public members of the base class become public members of the derived class and protected members of the base class become protected members of the derived class. A base class's private members are never accessible directly from a derived class, but can be accessed through calls to the public and protected members of the base class.

Protected Inheritance: When deriving from a protected base class, public and protected members of the base class become protected members of the derived class.

Private Inheritance: When deriving from a private base class, public and protected members of the base class become private members of the derived class.

Function Overriding:

- member function of base class is redefine in a derived class, then that function is called overridden. and this mechanism is called Function Overriding
- Function that is redefined must have same name, same return type and same parameter list in both base and derived class.
- Function overriding cannot be done within a class. For this we require a derived class and a base class
- we are calling the overriden function using Base class and Derived class object. Base class object will call base version of the function and derived class's object will call the derived version of the function.
- **Example**

```
#include<iostream.h>
#include<conio.h>
class A
{
```

```
        public:
        void show()
        {
        cout << "class  A";
        }
};
class B:public A
{
        public:
        void show()
         {
        cout << " Class  B";
        }
};
void  main()
{
A ab;
B bc;     //Derived class object

ab.show();
bc.show();
getch();
}
```

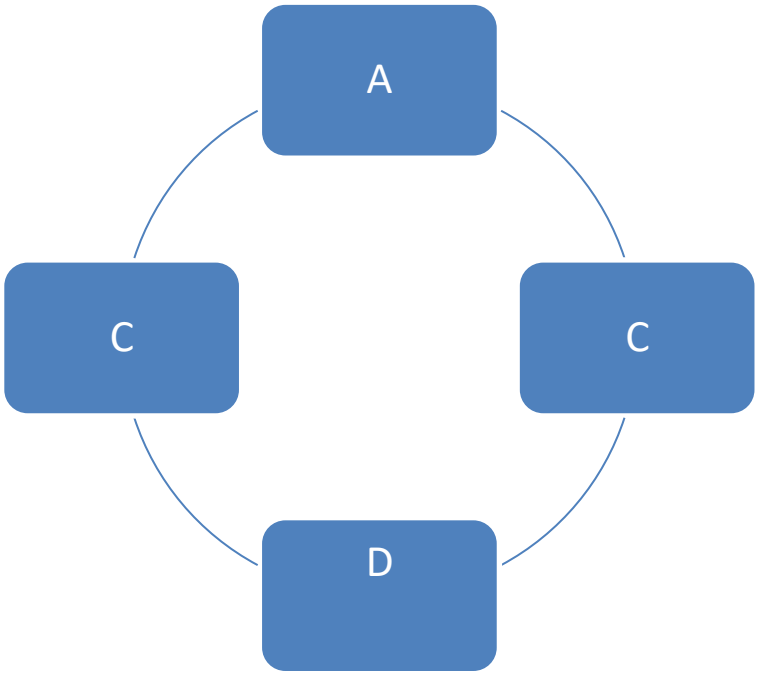
OUTPUT

```
Class  A

Class  B
```

Explain Virtual base class

- As shown in fig the class D is inherited from two base classes B and C. The class A is common base class for both B and C.

- Thus class B and C are parent classes for D, while class A is grandparent class.



- As you seen from fig, B and C both get properties of A . class D not only get the properties of its parent classes B and C, but also get the properties from its grandparent class A indirectly through parent classes.
- Due to this reason , it receives two copies of data defined in A ; one copy through class B and second through C
- This creates the duplication of data of class A in class D.
- To avoid this situation and inherit only one copy, C++ use the virtual base class.
- In our example , we have to declare the original base class means class A the virtual base class

Class A{};

Class B : **virtual** public A{};

Class C : **virtual** public A{};

Class D : public B, public C{};

- When a class is made as a virtual base class , C++ takes necessary care to see that only one copy of that class is inherited

- **Example of virtual base class**

```
class A
{
    public:
            void showA()
            {
            cout<<"class A";
            }
};
class B : virtual public A
{
    public:
            void showB()
            {
            cout<<"class B";
            }
};
class C : virtual public A
{
        public:
            void showC()
            {
            cout<<"class C";
            }
};
class D : public B, public C
{
```

```
        public:
        void showD( )
            {
            cout<<"class D";
            }
    };
void main()
{
        D cd;
        cd.showA();
        cd.showB();
        cd.showC();
        cd.showD();
        getch();
}
```

OUTPUT

Class a

Class b

Class c

Class d

Explain 'this' keyword

- this pointer is a one type of pointer which store the address of the object.
- in C++ , whenever a member function is called using object, the address of object is stored in **this** pointer and during execution of body.
- this pointer is used to access the data members.
- a this pointer is automatically passed to function when it is used..

Application of this pointer:

1) using this pointer, any member function can find out the address of the object of which it is member.
2) accessing data member with this pointer.
3) this pointer is to return the object it points to.

program of use of this pointer:

```
#include<iostream.h>
#include<conio.h>
class A
{
        public:
        void show()
        {
                cout<<"Address of Object ";
                cout<<this;
        }
};

void main( )
{
A ab,cd;
ab.show();
cd.show();
```

}

Virtual function with example

- Member function of a base class is redefined in a derived class. now we create a base class pointer and assign address of base class object to it ,Then it will call a base class function
- But If we assign address of derived class object to base class pointer ,Then it will call a base class function , not call a derived class function
- Because C++ goes by type of pointer means pointer type is base type, it calls base class function.
- By using virtual function , we can call derived class function through base class pointer
- A function is said to be virtual when same function name is used in both base and derived classes, and the keyword virtual is written before the return type of the function in the base class function
- A virtual function is a member function that is declared as virtual in a base class and redefined by a derived class.
- By using virtual function we can achieved run time polymorphism or late(or dynamic) binding
- In Run time polymorphism selection of appropriate function are called at run time
- **Virtual function characteristics:**
    - ✓ it must be member of some class
    - ✓ it can not be a static member
    - ✓ it can only be accessed by pointer
    - ✓ it must be defined in the base class
    - ✓ the prototype of the base class version of a virtual function and all derived class version must be same
    - ✓ a virtual function can be friend of another class

Consider following program code:

```
class A
 {
```

```
public:
        virtual void show()
        {
                cout <<"base class"<<endl;
        }
 };

Class B: public A
 {

        public:

        void show()
        {
                cout <<"derived class"<<endl;
        }
 };
void main()
 {
          A *p, ab;
          B cd;

        p = &ab;

        p->show();
         p = &cd;

          p->show();
getch();

 }
```

Output:

Base class
Derived class

Abstract base class

- A class having at least one pure virtual function is known as abstract class
- We can not create objects of abstract class as it is only used act as base for the derived classes in inheritance
- Anybody who tries to create object from such abstract base class would be reported an error by the compiler .
- Pure virtual Functions are virtual functions with no definition. They start with virtual keyword and ends with = 0.
- For example:

```
class base
{
    Public:
    virtual void show=0;
};
```

- in above example, the show() function is known as Pure virtual Function
- in abstract class you can create pointer and references. This allow abstract classes to support run time polymorphism
- example of abstract base class

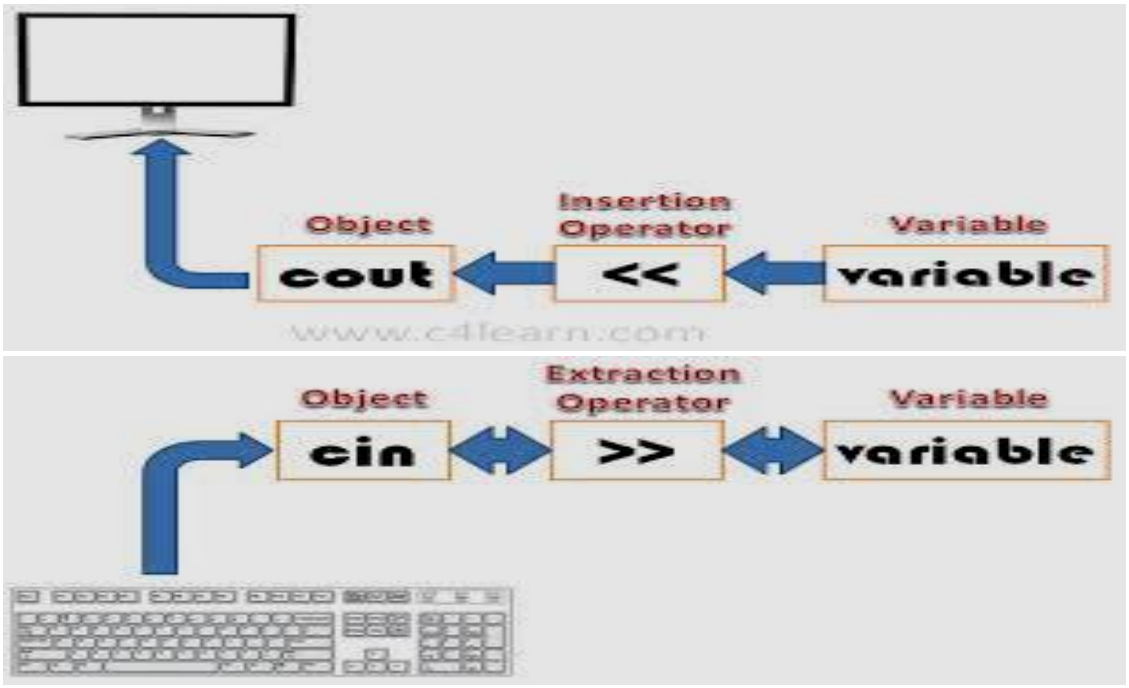
```
class A        //Abstract base class
{
 public:
 virtual void show() = 0;          //Pure Virtual Function
};

class B: public A
{
 public:
 void show()
```

```
 {
cout << "Implementation of Virtual Function in Derived class"; }
};
void main()
{
 A obj;       //Compile Time Error
 A *b;
 B d;
 b = &d;
 b->show();
getch();
}
```

Output:

```
 Implementation of Virtual Function in Derived class"
```

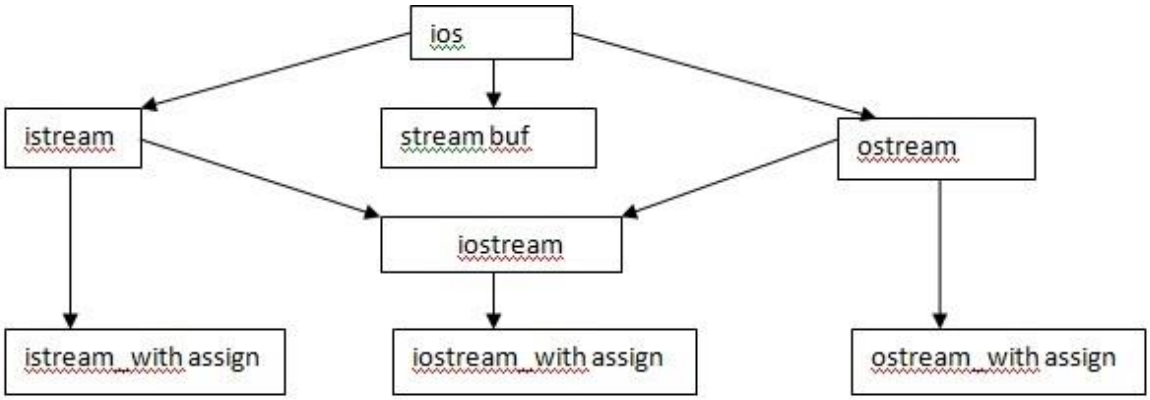
What is stream ? Explain c++ stream classes with a diagram



- A stream is collection of bytes.
- If the stream is provided from the input device to a program then it is called input stream. while sending the data from program to output device is known as output stream.
- Normally input device is keyboard while output device is display device
- C++ provides predefined streams whenever any program starts. They are **cin**, **cout**,**cerr**.
- The cin is by default connected to keyboard while cout and cerr are by default connected to monitor.
- We can use cin to receive the input from keyboard while cout is used to print the output on monitor.
- Fig show the use of cin with extraction operator >> for reading input from keyboard and cout to print output on screen using insertion operator

Explain Stream class hierarchy

- C++ stream classes are provided by the header file <iostream.h>. it provides the hierarchy of stream classes supporting variety of I/O operation.



- fig show the hierarchy of stream classes in C++
- As shown in fig the ios class is the top most class in hierarchy and act as base class for istream and ostream classes
- The ios class provides the basic facilities for other classes in hierarchy and defines number of I/O functions.
- The istream class is a input stream class and provides the input functions like, get(), getline(), read() etc. , and overloaded extraction operator >>. The cin is an object of istream class.
- The ostream class is a output stream class and provides the output functions like, put(), write() etc. , and overloaded insertion operator <<. The cout is an object of ostream class.
- To avoid the duplication in iostream class, the ios class is defined as virtual base class
- The ios class also contain a pointer to a buffer object called streambuf
- The istream_withassign, ostream_withassign, iostream_withassign classes add the assignment operator to the istream, ostream,iostream classes respectively.

Explain input/output unformatted function(get(),getline(),read(),put(),write())

- input stream class provides the input functions like, get(), getline(), read() etc. , and overloaded extraction operator >>. The cin is an object of istream class.
- output stream class provides the output functions like, put(), write() etc. , and overloaded insertion operator <<. The cout is an object of ostream class.
- The get() functions are used to read character and used to read an ASCII value of character.
- The put() functions are used to print character whose ASCII value is given as argument.
- Use of get() and put( )

```
char c;
cin.get(c);
// other way is
c=cin.get();

cout.put(c);
cout.put(65);
```

- the getline() function is used to read line of text containing with spaces.
- The general format of getline() is

  cin.getline(str,size);

  where **str** is character array(string)

  size means maximum number of characters it reads
- the read() function is used to read text with multiple lines.
- While entering the input in single or multiple lines for read() function whenever newline is pressed and if total number of character pressed including all line, exceeds the size, it will return with size characters stored in string.

- The general format of read() is

  cin.read(str,size);

  where **str** is character array(string)

  size means maximum number of characters it reads
- The write() function is used to display a line.
- The format of write function is

  cout.write(str,size);

  it display **size** characters from string **str**.

  - If the null character encounters before size, then also it continues to print the characters in case where total characters in string are less than size.

Formatted output with ios function and flags

- The ios class provides number of function for formatted output.
- The ios functions used for formatted output are in table

Function	Meaning
width( )	Specifies the width of the field for output
fill()	Specifies the fill character used to fill up the unused part of the field.
precision()	Specifies the number of digits to be displayed after decimal point in floating point numbers
setf()	Used to set the format flags to control the format of output
unsetf()	Used to reset the format flags

- **Flags with their bit-fields**

Flag	Bit-field	Meaning
ios::left	ios::adjustfield	Output is left justified
ios::right		Output is right justified
ios::inernal		Output is padded after sign or base indicator

ios::fixed	ios::floatfield	Floating point value is displayed in fixed format
ios::scientific		Floating point value is displayed in scientific format
ios::dec	ios::basefield	Base of the value is decimal
ios::oct		Base of the value is octal
ios::hex		Base of the value is hexadecimal
ios::showpoint		To display trailing decimal point and zero
ios::showpos		To display + sign before positive number
ios::showbase		To display base indicator

Program of use of manipulator:

```
#include<iostream.h>
void main()
{
float x=0.1234;
cout.width(10);
cout.fill('$');
cout.setf(ios::left);
cout<<70444;
cout.unsetf(ios::left, ios::adjustfield);
cout<<endl;

cout.precision(2);
cout<<x;

cout.width(10);
cout.fill('#');
cout.setf(ios::hex, ios::basefield);
cout.setf(ios::showbase);
cout<<256;
cout.unsetf(ios::hex, ios::basefield);
cout<<256

getch()
}
```

Output:
70444$$$$$
0.12
#####0x100
256

Explain manipulator.

- C++ provides the set of functions called manipulator for formatting the output.
- The <iomanip.h> header file provides manipulators for different formatting of outputs.
- Most commonly used manipulators are
    - ✓ **endl**
    - ✓ **setw(),**
    - ✓ **setfill()**
    - ✓ **setprecession()**
    - ✓ **setiosflags()**
    - ✓ **resetiosflags()**
- The **endl** is same the new line('\n') character .
- The **setw( )** is used to set the width of field for output.
  Example: **setw(x)** // x is field width
- The **setfill( )** is used after setw() manipulator. It is used to fill the unused part of field with a specific character.
  Example: **setfill('$')**
- The **setprecision( )** manipulator is used to set the number of digits to be displayed after decimal point in a floating point number. The by default value is **6**
- The **setiosflags( )** to set the format flags

Format flags are following

Flag	Meaning
ios::left	Output is left justified
ios::right	Output is right justified
ios::inernal	Output is padded after sign or base indicator
ios::fixed	Floating point value is displayed in fixed format
ios::scientific	Floating point value is displayed in scientific format
ios::dec	Base of the value is decimal
ios::oct	Base of the value is octal
ios::hex	Base of the value is hexadecimal
ios::showpoint	To display trailing decimal point and zero
ios::showpos	To display + sign before positive number
ios::showbase	To display base indicator

- The **resetiosflags()** is used to reset flags

Program of use of manipulator:

```
#include<iostream.h>
#include<iomainp.h>
void main()
{
float x=0.1234;
cout<<setw(10)<<setfill('$')<<setiosflags(ios::left)<<70444;
cout<<resetiosflags(ios::left)<<endl;
cout<<setprecision(2)<<x;
cout<<setw(10)<<setfill('#')<<setiosflags(ios::hex)<<256;
cout<<resetiosflags(ios::hex)<<endl;
cout<<256;
getch()
}
```

Output:

```
70444$$$$$
0.12
#####0x100
256
```

User defined manipulators

- It is possible to add user defined manipulators or formatted function.
- The general form of creating a manipulator without any argument is:

```
ostream & manipulator(ostream & output)
{
    ----------------------
    ----------------------
    Code come here
    return output;
}
```

- Example of user defined manipulator

```
#include<iostream.h>
ostream& abc(ostream & output)
{
    output<<"kgs";
    retrun output;
}
void main( )
{
cout<<10<<abc;
}
```

Output:

10kgs