Name -> shubham

C++ => Programming

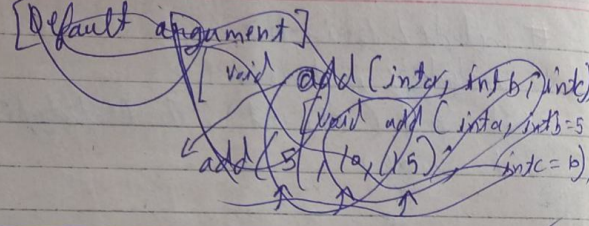
class => BCA III Sem

Chapter => Functions

C++ Functions

```
#include<conio.h>
#include<iostream.h>
void add(int, int); // Prototype
int main()
{
  int a, b;
  cout<<"\n Enter two numbers:";
  cin>>a>>b;
  add(a, b);
  getch();
  return 0;
}
void add(int a, int b)
{
  int c;
  c = a+b;
  cout<<"\n addition ="<<c;
}
```

[Default argument]



[Default Argument]

Prototype

Void add (int a, int b, int c);

Void add (int a, int b = 5, int c=10);

add (5, 10, 15);

① a = 5, b = 5, c = 10

1> add (5); ② a = 5, b = 10, c = 10

2> add (5, 10);

3> add (5, 10, 15); ③ a = 5, b = 10, c = 15

[Default Argument]

Void add [int a = 5, int b = 10, int c = 15);

{

① add () => a = 5 / b = 10 / c = 15

② add (20) => a = 20 / b = 10 / c = 15

③ add (10, 20) => a = 10 / b = 20 / c = 15

④ add (10, 20, 30) => a = 10 / b = 20 / c = 30

[Default Argument] => Default argument provides a way to give a default value to the argument. If we don't supply the argument, it takes the default value, otherwise the passed value.

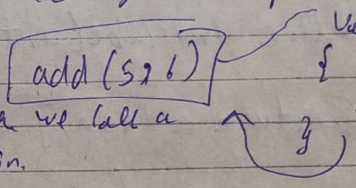
Rule once we have started giving default value, all next argument must contain a default value.

```
#include<iostream.h>
#include<conio.h>
void add(int a=5, int b=10);
int main()
{
  add(_);  -①
  add(10);
  add(10,20);
  getch();
  return 0;
}
  void add(int a, int b)
{  cout<<"Addition="<<a+b;
  }
  output:
   Addition =15
   Addition = 20
   Addition = 30
```

2

[Inline Function]

A function is used to reduce code redundancy as well as to save the memory space. when a function is invoked. a bunch of tasks is performed viz. matching arguments, matching return and passing the control from calling to definition & vice-versa whenever we call a function.



But when function definition consist of hardly one or two simple statements then this bunch of tasks appears to be time consuming. Hence in order to save this time, C++ has the concept of inline function.

when a function is declared inline the function body is replicated at function calling time

Inline function Demo Program

Program Execution

```
#include <iostream.h>
#include <conio.h>
inline int Square(int a)
{
return a*a;
}
inline int Cube(int a)
{
return a*a*a;
}
int main()
{
cout << "In Square of 5=" << Square(5);
cout << "In cube of 5 = " << Cube(5);
getch();
return 0;
}
```

3) Function overloading :

Function overloading

Concept of polymorphism enables us to write same name multiple function / methods within a program.

But we have a restrictions that all the function having same name must follow.

Difference shall be in (1) Nb. of Arguments
(2) Type of Argument

Prototype :

```
void add(int, int);      [ Nb. of Arg = 2
void add(int, float);    [ Nb of Arg = 2
void add(int, int, int); [ had same
```

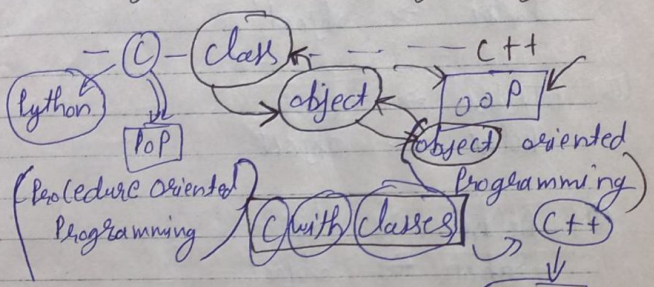
Call

```
Add(10, 20);
Add(10, 10.10);
Add(10, 20, 30);
```

```
# include <iostream.h>
# include <conio.h>
  void add (int, int);
  void add (int, int, int);
  void add (int, float);
    int main ()
  {
    add (2, 5);
    add (10, 20.10);
    add (10, 20, 30);
     getch ();
    return 0;
   }
   void add (int a, int b)
{
   cout << "In Addition = " << a+b;
  }
  void add (int a, float b)
  {
    cout << "In Addition = " << a+b;
  }
  void add (int a, int b, int c)
{ cout << "In Addition = " << a+b+c; }
```

C++ PROGRAMMING

object oriented Programming (OOP)



It is the combination of data members & member function put together into a single Unit

object ⇒ object is an instance of a class

class ⇒ object

① Universe → class
Milky-way Galaxy → object
② Milky-way Galaxy → class
Earth → object

③ Earth → Class
India → object

④ India → class
Delhi → object

⑤ Delhi → class
Sabet → object

⑥ Sabet → class
Home → object

class → user define data type
class-name
{
- - -
- - -
- - -
}
object

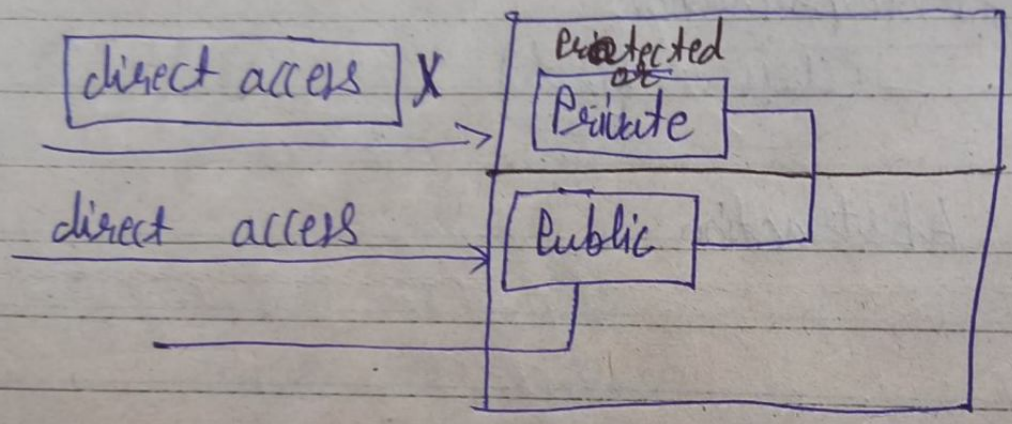
① class & object
② Inheritance
③ function overloading } → Polymorphism
④ constructor overloading }
⑤ operator overloading
⑥ Encapsulation
class & object
⑦ Abstraction

study deep in Inheritance

Access specifier / visibility modifiers

(C++) (data restriction)

1. Private → [directly] – X
2. Protected → [Inheritance]
3. Public → directly ✓



[object oriented Programming]

⇓

Class & object

⇓

Combination of
data Member (variable)
+
Member function (function)

Programing with class & object

- Private
- Protected
- Public

Access specifier

Private
↳ Variable
Data Members

Public
function
Members function

// Program to add two numbers using class & object.

```
#include<iostream.h>
#include<conio.h>
class addition
{
    Private:
    int a, b;
    Public:
    void getdata();
{
    cout<<"\n Enter Two Numbers:";
    cin>>a>>b;
    }
    Void Putdata();
    {
    cout<<"\n Addition="<<a+b;
    }
    };
int main()
{
    addition aa;
    aa.getdata();
    aa.Puddata();
    getch()
    return 0;
    }
```

aa

Private
a 5, b 6
Public
getdata()
Putdata()

// Program to check number odd even using Class & object

```
#include <iostream.h>
#include <conio.h>
class oddeven
{
  Private:
    int a;
  Public:
    Void getdata();
    Void Putdata();
};
Void oddeven :: getdata()
{
  Cout << "\n Enter No to check:";
  Cin >> a;
}
Void oddeven :: putdata()
{ if (a%2 == 0)
  Cout << "\n Even";
  else
  Cout << "\n odd;
}
```

class

object

```
int main()
{
  oddeven aa;
  aa.getdata();
  aa.putdata();
  getch();
  return 0;
}
```

// Program to Print 1 to 10 numbers using class & obj.

```
#include <iostream>
#include <conio.h>
class oneten
{
  Void putdata()
  {
  int i;
  for (i=1; i<=10; i++)
    Cout << "\n" << i;
  }
};
int main()
{
  oneten aa;
  aa.putdata();
  getch();
  return 0;
}
```

Program to find reverse of a number using class and object.

Programming with class & object

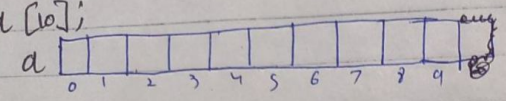
```
#include <iostream.h>
#include <conio.h>
  class reverse
{
    int n;
    public:
      void getdata();
      void putdata();
    };
    void reverse :: getdata()
    {
      cout << "\n Enter the No to reverse:";
       cin >> n;
     }
    void reverse :: putdata()
{
    int rev=0
    while(n>0)
  { rev = (rev*10) + n%10;
    n = n/10; }
  cout << "\n Reverse = " << rev;
  }
```

logic for Rever a number

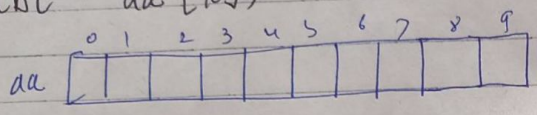
```
int main()
{
  reverse aa;
  aa.getdata();
  aa.putdata();
  getch();
  return 0;
}
```

Array of object

int a[10];



abc aa[10];



object of array

Class xyz

{ int a[10]

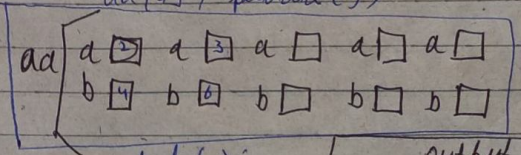
};

xyz aa;

aa[a

Array of object

#include<iostream.h>
#include<conio.h>
class xyz
{
int a, b;
Public:
void getdata();
{
cout<<"\n Enter Two numbers";
cin>>a>>b;
}
void putdata()
{ cout<<"\na="<<a<<"\tb="<<b;
}
};
int main()
{
xyz aa[5];
int i;
for(i=0;i<5;;i++)
aa[i].getdata();
for(i=0;i<5;i++)
aa[i].putdata();



dry run

getch();
return 0;
}

output

Program to add Time using class

```
#include<iostream.h>
#include<conio.h>
  class Time
{
   int h, m;
   Public :
    void getdata()
   {
    Cout<<"In Enter Hours & Minutes:";
    Cin >> h >> m;
   }
    void Putdata()
   {
    Cout<<"In Hours = "<< h;
    Cout <<"\t Minutes = "<< m;
   }
    void sum(Time T1, Time T2)
  {
    h = (T1.m + T2.m)/60;
    m = (T1.m + T2.m)%60
    h = h + (T1.h + T2.h);
   }
    };
```

Dry run

object as [function Argument]

getdata	T_1	h 2	m 40
getdata	T_2	h 3	m 50
sum(T_1, T_2)	T_3	h 6	m 30

```
int main()
{
  Time T1 , T2 , T3 ;
  T1.getdata(); T2.getdata();
  T3 . sum( T1 , T2 );
  T1 . Putdata(); T2.Putdata();
   T3 Putdata();
   getch(); return 0;
}
```

Friend Function

we know that the private section of class is accessable only & only through the Public Section of the same class

what if, we want to give Access to private member, a function outside the class. In such circumstances we use the Concept of friend function.

Key point

1) A friend function can't be called using the object of the class. It is called like a normal function.
2) Friend function can use the resources of a class only using an object of the same class
3) Usually a friend function has object as an argument.

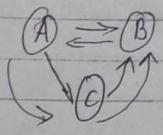
```
class abc
{ int a, b; Public:
    ;; };
abc aa
function / friend
```

```
#include<iostream.h>
#include<conio.h>
class demo
{
  int a, b;
  Public:
    void getdata();
    friend int Sum(demo);
};
void demo::getdata()
{
  cout<<"\n Enter two Number:";
  cin>>a>>b;
}
int Sum(demo aa)
{
  return(aa.a + aa.b);
}
int main()
{
  demo aa;
  aa.getdata();
  cout<<" Addition=" <<Sum(aa);
  getch(); return 0; }
```

aa { a 5 b 2 Public: getdata() }

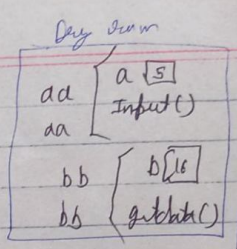
output: Addition=7

Friend function



class A
{ x
};
class B
{ y
};

```
#include<iostream.h>
#include<conio.h>
 class B;
 class A
{
    int a;
   Public:
    void input()
{
    cout<<"\n Enter No:";
    cin>>a;
    }
    friend void max (A, B);
};
```




```
class B
{
  int b;
  Public
   void getdata()
{  cout
    cout<<"\n Enter No:";
    cin>>b;
   }
    friend void max(A, B);
};
 void max (A aa , B bb)
{
  if (aa.a > bb.b)
     cout<<"\n Max="<<aa.a;
  else
     cout<<"\n Max= "<<bb.b;
}
```

```
int main()
{
  A aa; B bb;
aa.input();
bb.getdata();
max(aa, bb);
 getch();
 return 0;
}
```

[Static Data Member]

1) It is initialized to zero, whenever the first object of its class is created No other initialization is permitted.

2) for making any data member static we use "static" keyword.

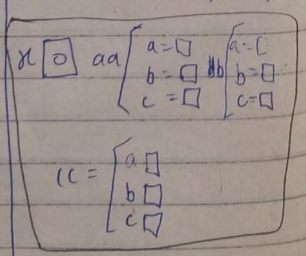
3) only one copy of static data member is created & shared by all

4) It's visibility is entire program.

Data member

```
class demo
{
  static int x;
  int a, b;
  Public:
  int c;
  void getdata();
  void Putdata();
};
```

```
demo aa, bb, cc;
```



getdata()
Putdata()

```
#include<iostream.h>
#include<conio.h>
class demo
{ int x, y;
  static int z;
  Public:
  void getdata(int a, int b)
  {
    x = a;
    y = b;
    z = z+1;
  }
  void Putdata()
  { cout<<"\n x="<<x<<"\t y="<<y<<"\t z="<<z;
  }
};
int demo::z;
void main()
{
  demo aa, bb; aa.getdata(5,10);
  bb.getdata(20,30); aa.Putdata();
  bb.Putdata();
  getch(); }
```

output		
x=5	y=10	z=2
x=20	y=30	z=2

Static Member function

① Static member function can access only static data members

② Since it is not the part of any object, it is call using the class name.

class name

3 □

Static void myfunc()

Static member function

class XYZ

{

int a, b;

Static int Z;

Public:

void getdata();

void putdata();

Static void myfunc();

};

XYZ aa, bb;

XYZ :: myfunc();

data member

Static data member

aa
- a □
- b □
- getdata()
- Putdata()

bb
- a □
- b □
- get data()
- Put data()

```
#include<iostream.h>
#include<conio.h>
class demo
{
  int x;
  Static int y
  Public:
  Void getdata (int a)
  {
    x = a;
    y = y+1;
  }
  Void Putdata()
  { cout << "In x =" << x << "In y =" << y;
  }
  Static Void abc()
  {
    cout << "In y =" << y;
  };
  int demo :: y;
```

```
int main()
{
  demo aa;
  aa.getdata(10);
  aa.Putdata();
  demo::abc();
  getch();
  return 0;
}
```

chapter Constructors

⇒ Constructors are special member function which is used to initialize value of a variable inside an object.

The major points about Constructors:-

1) A Constructor's name is same as the class name :-
2) A Constructor is automatically invoked as same as an object of its class is created.
3) A Constructor hasn't any return type not Even "Void"
4) Constructor allows default argument concept.
5) A Constructor can't be inherited

int a=5;
a [5]
Built in Datatype
C++ => class & object
user defined datatype

abc aa, bb, ;
aa. input ();

Constructors

1) Default Constructor
2) Parameterized Constructor
3) Copy Constructor.

1) Default Constructors

Default Constructor is a type of ~~Comp~~ Constructor whose name is similar to the class name & which is automatically invoked.

A Constructor having no argument

```
class abc
{
  int a;
  Public:
  -> abc()
  {
    a=10;
  -> }
  };
```

```
class abc
{
  int a;
  Public:
    abc();
  };
  abc :: abc()
  {
    a=10;
  }
```

```
#include<iostream>
#include<conio.h>
  class demo
{
   int a, b;
   Public
     demo()
   {
     a=10;
     b=20;
   }
     Void Putdata();
};
  Void demo::Putdata()
{
  Cout<<"a="<<a<<"\t b="<<b;
}

  int main()
{
   demo aa;
   aa.Putdata();
   demo bb;
   bb.Putdata();
   return 0;
   getch();
}
```

aa [a[10] b[20]

output

a=10 b=20

bb [a[10] b[20]

output

a=0 b=20

2) Parameterized Constructor

A Constructor is a special member function whose name is same as the class name & which is automatically invoked as soon as an object of its class is created.

A Constructor having argument is called parameterized Constructors.

Syntax:

```
class demo
{
    int a, b;
    public:
    demo (int m, int n)
    {
        a = m;
        b = n;
    }
};
```

```
demo aa(10,20)
demo aa();
```

```
#include<iostream.h>
#include<conio.h>
class demo
{
    int a, b;
    public:
    demo(int m, int n)
    {
        a = m;
        b = n;
    }
    void Putdata()
    {
    cout<<"a="<<a<<"\n b="<<b;
    }

    int main()
    {
    demo aa(5, 10);
    aa.Putdata();
    getch();
    return 0;
    }
```

aa → a 5, b 10

output
a=5
b=10

or

```
int main()
{
 int x, y;
cout<<"\n Enter two";
 cin>>x>>y;
 demo aa(x, y);
 aa.Putdata();
 getch();
 }
```

3) Copy Constructure

Constructors are special member function whose name is same as class name and which is automaticaly invoked when an object of its class is created.

when we need to initialize the variable of an object with the values of variables of another object of same type, Then we use the concept of Copy Constructor.

Example:

```
demo aa;
demo bb = aa;
  or
demo bb(aa);
```

aa [a 5 b 10]

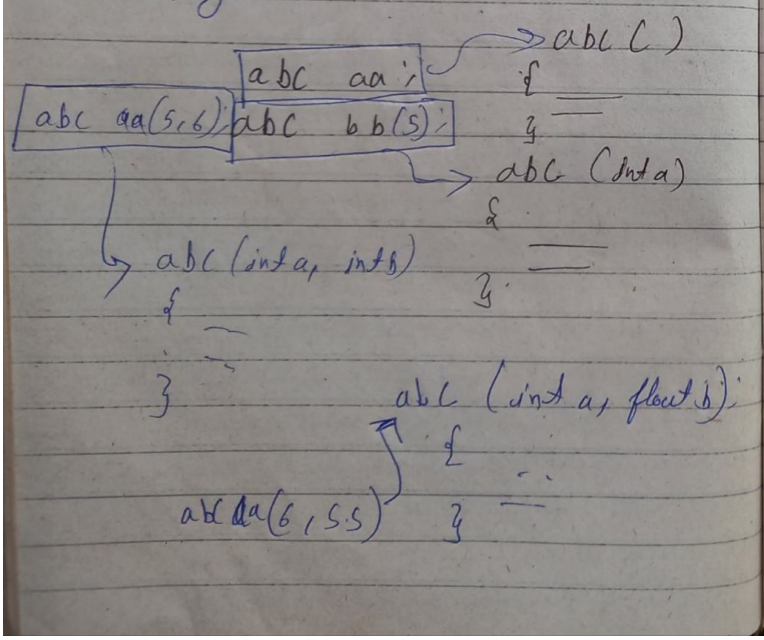
bb [a 5 b 10]

```
#include<iostream.h>
#include<conio.h>
 class demo
{
  int a;
  Public:
   demo()
  {
   a=10;
  }
  demo(demo &z)
{
   a = z.a;
  }
   Void Putdata()
{
   Cout << "\n A = " << a;
  }
};
     int main()
   {
     demo aa;
     demo bb(aa);
  or demo cc = bb;
```

```
aa.Putdata();
bb.Putdata();
cc.Putdata();
return 0;
}
```

Constructor overloading

C++ provides us the provision using which we can incorporate more than one constructors in a single program. But these constructors must have different types of argument / different no. of arguments. This provision of having more than one constructors in a single program is called constructor overloading.




```
#include<iostream.h>
#include<conio.h>
class demo
{
  int a;
  public:
  demo()  // default constructor
  {
    a=10;
  }
  demo(int x)  // parameterized constructor
  {
    a=x;
  }
  demo(demo &z)  // copy constructor
  {
    a=z.a;
  }
  void putdata()
  {
    cout<<"\n A="<<a;
  }
};
```

```
int main()
{
  demo aa;      aa[a 10]
  demo bb(20);  bb[a 20]
  demo cc(aa);  cc[a 10]
  aa.putdata();
  bb.putdata();
  cc.putdata(); }
  return 0;}
```

Destructors

Like Constructors, destructors are also member function whose name is similar to the class name having (~) tilled sign just before its name.

A destructor is used to destroy object once the object goes out of scope.

A destructor has no return type as well as it doesn't take any argument.

```
class A
{
  ---
  A()
  {
  }
};
```

Constructors	Destructor
▷ Values Initialized	Destroy object memory
▷ Argument ✓	Argument ✗
▷ return type ✗	return Type ✗
▷ It's name is similar to the class name	Its name is similar to the class / tilled Sign Preceded by

```
class A
{
  ---
  A()
  {
  }
  ~A()
  {
  }
};
```

Destructors

count = 0 1 2 3

1. aa
2. bb
3. cc
4. dd

output:
no of object created 1
no of object create 2
" " " 3
" " " 4
" destroy 3

```
#include<iostream.h>
int count=0;
class demo
{
public:
  demo()
  {
    count++;
    cout<<"In No. of object Created"
        <<count;
  }
  ~demo()
  {
    count--;
    cout<<"\n\tNo. of object destroyed"<<co
  }
};
int main()
{
  demo aa, bb, cc;
  {
    demo dd;
```

count = 3 2 1 0

no of object dest
" " " "
" " " dest 0

- Inheritance -

Inheritance is one of the most important feature of object oriented programming. This provides us the capability to re-use the pre-existing code to our new project.

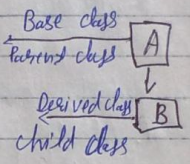
OOP - Reusability Properties
1) class & object
2) Inheritance

It is always better to re-use the pre-existing code rather than developing all over again. This re-usability feature is incorporated by the concept of Inheritance.

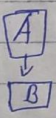
Types of Inheritance
1) Single Inheritance
2) multiple "
3) Multi-level "
4) Hierarchical "
5) Hybrid Inheritance

1) Single Inheritance

The type of Inheritance in which one class is being inherited by another class.



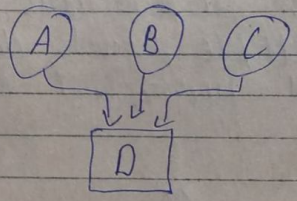
example



2) Multiple Inheritance

The type of Inheritance in which more then one class is being inherited by another class.

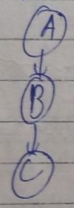
Example



3) Multilevel Inheritance:

The type of Inheritance in which one class inherits other class, and then this class is being inherited by another then this is called multilevel Inheritance

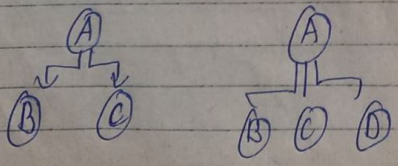
Example



4) Hierarchical Inheritance:

The type of Inheritance in which one base class is being inherited by multiple derived class then it is called Hierarchical Inheritance.

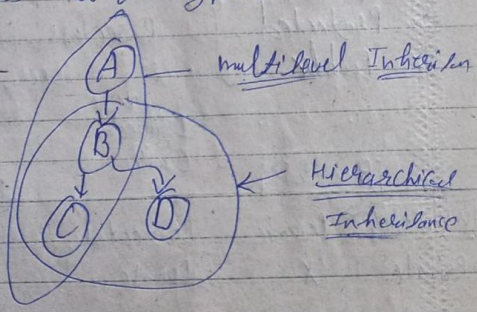
Example



5) Hybrid Inheritance

when we combine two or more types of inheritance into one, then it is called Hybrid Inheritance.

For ex:-



Defining Derived classes

class & object

Access specifier

1) Private
2) Protected
3) Public

⇒ [Inheritance] [visibility modifiers]

Baseclass → Derived class

Inherit by → Private, Protected, Public

Normal Program

Base class / V.V. Inherit	Private	Protected	Public
1) Private	X	Private	Private
2) Protected	X	Protected	protected
3) Public	X	protected	Public

1) class abc

Private: a, b
Protected: c, d
Public:
getdata()
Putdata()

class xyz : private abc

Private: ~~a, b~~ c, d
getdata
Protected: o
Public:
Input

2) class abc

Private: a, b
Protected: c, d
Public:
getdata()
Putdata()

class xyz : protected abc

private: x, y
Protected: o
c, d
Putdata(), getdata()
Public:
input(), output()

3) class abc

Private: a, b
Protected: c, d
Public:
getdata()
Putdata()

class xyz : ~~protected~~ public abc

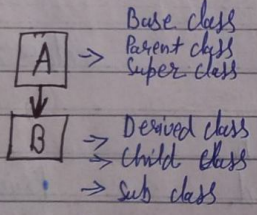
Private: m, n
Protected: o
c, d
Public:
Input(), output()
getdata()
Putdata()

-Single Inheritance-

when one base class is being derived by a single sub-class, then this is called single inheritance.

It means when there is one base class & it is being derived by another single derived class then it is called single Inheritance.

For example :




```
#include<iostream.h>
#include<conio.h>
  class A
{
  Protected:
   int x;
  Public:
   void input()
  { cout<<"In Enter Number";
```

A { Protected : x
Public : input()

```
class B: Public A
{
  int y;
  Public:
   void getdata()
{
   cout<<"In Enter No.";
   cin>>y;
}
   Void putdata()
{ cout<<"In Addition="<<x+y;
}};

int main()
{
   B aa;
aa.input();
aa.getdata();
aa.putdata();
getch();
return 0;
}
```

B { Private : y 5
Protected : x 10
Public : getdata(), Putdata(), input()

output { Enter number 5
Enter number 10
Addition = 15

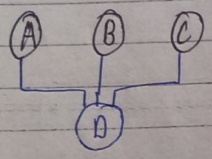
```
#include<iostream.h>
#include<conio.h>
class A
{
  int x;
 Public:
  void input()
 {
   cout<<"\n Enter number";
    cin>>x;
  }
    int getx()
    { return x;
      }
  };
class B: Public A
{
   int y
   Public:
    void getdata()
   {
     cout<<"\n Enter No";
     cin>>y;
    }
```

```
  void Putdata()
 {
   cout<<"\n Addition="<<getx()+y;
  }
  };
  int main()
 {  B aa;
  aa.input();
  aa.getdata();
  aa.Putdata();
  getch();
  return 0;
 }
```

- Multiple Inheritance -

When there are more than one base class is being inherited by a single derived class, then it is called the concept of multiple inheritance.

for ex:-




```
#include<iostream>
#include<conio.h>
class A
{
  Protected:
    int a;
  Public:
    void input()
    {
      Cout<<"\n Enter number:";
      Cin>>a;
    }
};
```

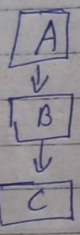
```
class B
{
  Protected:
    int b;
  Public:
    void getdata()
    {
      Cout<<"\n Enter No:";
      cin>>b;
    }
};

class C: Public A, Public B
{
  Public:
    Void addition()
    {
      Cout<<"\n Addition="<<a+b;
    }
};
int main()
{ C aa; aa.input(); aa.getdata(); aa.addition
  getch(); return 0; }
```

- Multilevel Inheritance -

Multilevel inheritance is a type of inheritance in which one super class/base class is derived by a child class & then this child class is further derived by another child class & so on. This type of inheritance is called multilevel inheritance.

for example →




```
#include<iostream.h>
#include<conio.h>
class A
{ protected:
  int roll;
  public:
   void getroll()
   {
    cout<<"\n Enter Roll";
    cin >> roll;
   }
   void putroll()
   {
    cout<<"\n Roll No = "<< roll;
   }
};
class B : public A
{
  protected:
  int sub1, sub2;
  public:
   void getmarks()
   { cout <<"\n Enter marks of 2 sub: ";
     cin >> sub1 >> sub2;
   }
   void putmarks()
  { cout <<"\n Marks 1 = "<< sub1;
    cout <<"\n marks 2 = "<< sub2;
  }
};
class c : public B
{  int sptm;
   public:
    void getsptm()
   {
```

```
    cout<<"\n Enter sports marks: ";
    cin >> sptm;
  }
  void total()
{
    getall();
    getmarks();
    cout<<" spt marks = "<< sptm;
    cout<<"\n total marks = " << sub1+sub2+sptm;
  }
}
int main()
{
    C aa;
  aa.getall();
  aa.getmarks();
  aa.getstm();
  aa.total();
   getch;
  return 0;
}
```

— Hierachical Inheritance —

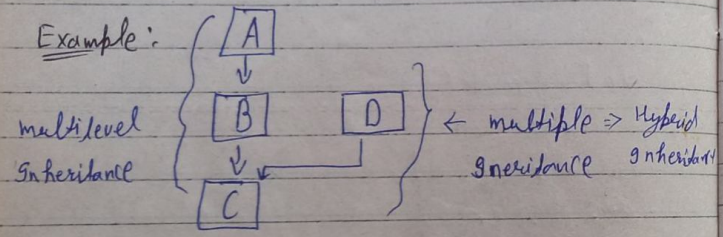
when one base class is being inherited by multiple derived class. Then this is called the Concept of Hierarchical Inheritance.

```
#include<iostream.h>
class A
{
  Public:
   Void message()
  {
    Cout<<"In welcome Inheritance:";
   }
};
class B : Public A
{
   Public:
    Void display()
   {
     Cout<<"In Inside class B:";
    }
};
class C : Public A
{
   Public:
    Void putdata()
   {
     Cout<<"In Inside class C:";
   }
};
   int  main()
{
   B aa; C bb;
   aa.display(); aa.message();
   bb.putdata(); bb.message();
   return 0;
}
```

- Hybrid Inheritance -

when we combine the concept of two or more basic inheritance type (single inheritance, multiple inheritance, multilevel inheritance, Hierarchical inheritance) then the new type of inheritance is reffered to as Hybrid inheritance.

Example:




```
#include<iostream.h>
class A
{
  Public:
    void putdata()
    {
      cout<<"In Inside class A:";
    }
};
```

```
class B: Public A
{
  Public:
    void display()
    {
      cout<<"In Inside class B:";
    }
};
class C
{
  Public:
    void message()
    {
      cout<<"In Inside class C";
    }
};
class D: Public B, Public C
{
    Public:
      void print()
      { cout<<"In Inside class D";
      }
};
```

```
int main()
{ D dd;
  dd.print();
  dd.display();
  dd.message();
  dd.putdata();
}
```

Function Overriding

Suppose we have two class having same name function & also having same Signature, so in this case, obviously object of derived class is created & when calling that function. the function of derived class is called & executed. So here, the function of derived class has overriden the function of base class. This concept is called Method overriding / function overriding.

```
#include <iostream.h>
class A
{
  Public:
    void display()
    {
      Cout << "In Base class";
    }
};
```

```
class B: Public A
{
  Public:
    void display()
    {
      Cout << "In Derived Class"; A:: display();
    }
};
  int main()
{
    B aa;
    aa.display();
    return 0;
}
```

Or

```
class B: Public A
{ Public:
    void display(){
    Cout << "In Derived class"; }
};
```

```
int main()
{ B aa;
  aa.display();
  aa.A::display();
```

Constructor Handling

```
class A
{
  A()
  {
  }
};
```

```
class B : public A
{
};
int main()
{
  B aa;
}
```

```
#include <iostream.h>
class A
{
protected:
  int a;
public:
  A(int x)
  {
    a = x;
  }
  void display()
  {
    cout << "In A = " << a;
  }
};
```

```
class B
{
protected:
  int b;
public:
  B(int y)
  {
    b = y;
  }
  void putdata()
  {
    cout << "In B = " << b;
  }
};
```

```
class C : public A, public B
{
  int c;
public:
  C(int p, int q, int r) : A(p), B(q)
  {
    c = r;
  }
  void show()
  {
    cout << "In C = " << c;
  }
};

int main()
{
  C aa(10, 20, 30);
  aa.display(); aa.putdata(); aa.show();
  return 0;
}
```

Ambiguity Resolution

in Inheritance

```
#include <iostream.h>
class A
{
  Protected:
     int a;
  Public:
    void input()
  {
    Cout<<"\n Enter Value:";
    Cin>>a;
  }
    void output()
  {
    Cout<<"\n a = "<<a
  }
};

class B
{
  Protected:
    int b;
  Public:
      void input()
{
  Cout<<"\n Enter value:";
  Cin >> b;
  }
  void putdata()
{
  Cout<<"\n b = "<<b;
}
};

class C : Public A, Public B
{
  int c;
  Public:
    void input()  / void getdata()
  {
    Cout<<"\n Enter value:";
    Cin>>c;
    A::input(); B::input();
  }
    void display()
  {
    Cout<<"\n c = "<<c;
  }
};

int main()
{
   C aa;
   aa. Input();/aa.getdata();
   aa.display(); aa.output(); aa.putdata();
   return 0; }
```

- Virtual Function -

1) Early Binding

```
#include<iostream.h>
class A
{
  public:
     void show()
     {
       cout<<"In Base Class";
     }
};
class B: public A
{
  public:
     void show()
     {
       cout<<"In Derived class";
     }
};
int main()
{
  B aa; aa.show(); aa.A::show();
  return 0;}
```

2) Late Binding / Virtual function

```
#include<iostream.h>
class A
{ public:
  virtual void show()
     {
       cout<<"In Base class";
};
  class B: public A
{
    public:
      void show()
      { cout<<"In Derived class"; }
};  int main()
{  A *bptr; B aa;
   bptr=&aaa;
   bptr->show(); return 0;
  }
```

Virtual function =>

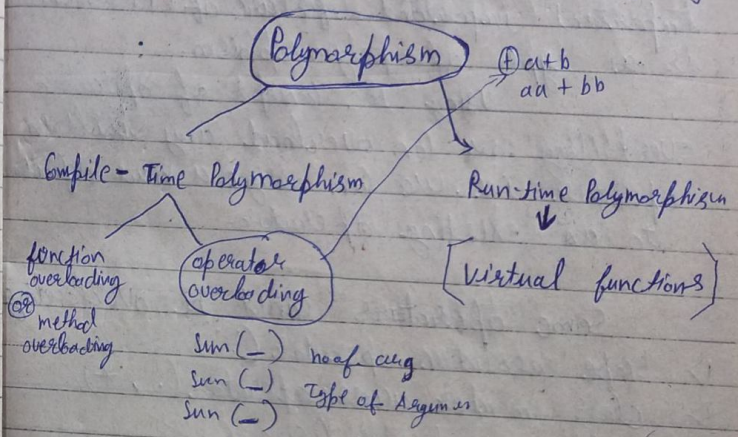
(*) virtual is a keyword in C++.

(*) A virtual function is redefined in derived class

(*) when a virtual function is defined in base class, then the pointer to base class is created. Now, on the basis of type of object assigned the respective class function is called

Polymorphism

It is one of the most important feature of OOP which simply implies [one name multiple form]



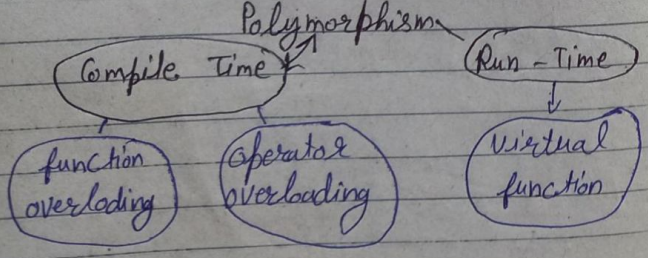
operator Overloading

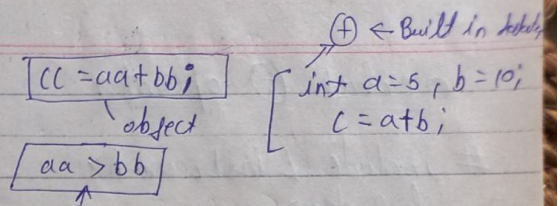
operator overloading is one of the most important important feature of C++ which is a type of polymorphism.

Using the concept of operator overloading, we can overload any built in operator i.e. we can assign a new definition to an existing operator.

Some operators can't be overloading

1) Scope resolution operator (::)
2) class member access operator (., *)
3) Size of operator (sizeof)
4) Conditional operator (? :)





Overload ⊕ operator

int a=5; b=10;
int c;
c = a+b;

a/b = Built-in datatype

demo aa, bb, cc;
cc = aa + bb
× Error

object
class ⇒ User defined data type.

overload (+) operator

```
# include <iostream.h>
class demo
{
   int a;
  Public:
     Void getdata()
    {
      Cout << "In Enter No.:";
      Cin >>a;
    }
      Void putdata()
    {
      Cout << " In Value = " <<a;
    }
  demo operator+ (demo bb)
{
     demo cc;
     cc.a = a + bb.a;
      return cc;
      }
  };
```

aa [a ☐
bb [a = ☐
cc [a = ☐

```
   cc = aa + bb;
```

function ↗ ↑ ↑
with argument with return

```
    int main()
{
    demo aa, bb, cc;
    aa. getdata();
     bb. getdata();
      cc = aa + bb;
     aa. putdata();
     bb. putdata();
      cc. putdata();
      return 0;
    }
```

when we overload (-) (/) (*) operator it is same as the overload (+) operator simply change the sign.

Overloading ++/-- operator

```
#include <iostream.h>
class demo
{
    int x;
    public:
     void getdata()
     {
Input  {  cout<<"\n Enter No: ";
       cin>>x;
     }
     void putdata()
Output {
     {
       cout<<x;
     }
```

Output: original = 5
Value after increment = 6

```
  void operator ++()
{
    x = x+1;
  }
};
```

```
int main()
{
  demo aa;
  aa.getdata()
  cout<<"\n original value = ";
   aa.putdata();
   ++aa; aa++;
  cout<<"\n Value after increment;
   aa.putdata();
  return 0;
}
```

(++) (--)

```
int a;
a = 5;
a = ++i;
cout<<a;
```

String operations

C++ has various built-in functions to perfor different operation on strings. this makes working with string more easier.

Functions	Description
length()	finds the length of a string
append()	joins two strings
getline()	takes entire line as string Input
swap()	swap values of two string variable

```
int main() {
    string str1 = "C++ Yogesh";
    cout << "String" << str1 << endl;
    // Str length
```

- `cout << " str1.length(); << endl;`

```
string str2 = "Hello";
    // join two strings
str3 = str1.append(str2);
```

- `cout << "joined string:" << str3 << endl;`

```
// get entire line as string Input
    getline(cin, str);
    cout << "String =" << str;
```

Swap two string values

```
string str1 = "C++";
string str2 = "Python";

str1.swap(str2);
cout << str1;          // Python
cout << str2;          // C++
```

More on String

1) String using char Array

In C++ we can also create string as an array of char types for examples.

Ex-1 char str[] = "C++";

Here, str is a string it includes "4" characters 'C', '+', '+' and '\0' at the end

Ex-2 char str[] = {'C', '+', '+', '\0'};

Example :- String as char array

```
int main() {
char str[] = {'C', '+', '+', '\0'};
Cout << str
```

Output => C++

=> Cout char string Input

```
int main() {
  char str[100];
  Cout << "Enter a word: ";
  Cin >> str;
```

output // enter a word : Gold
↓
Gold

it will only print a single word to take multiple words, replace :-

Cin >> str;

by

Cin.get (str, 100);

Here, 100 is the maximum number of characters we stored in it.

String library Functions

In C++, there is a string header file called cstring. This header file provides various library functions to perform operation on C-strings.

	Function	
1)	strlen()	Find the length of C-string
2)	strcat()	append copy of a C-string and entire another C-string
3)	strcpy()	copies a C-string to another

** toupper is used to convert lower to upper case.

To use header file in our program, we must first include cstring Header file in our program.

```
#include<cstring>
```

1. Strlen() : find length of String.

```
char str[100] = "C++ Programming";
cout << strlen(str);
```

output // 15

2) Strcat(); join two String

```
char str1[] = "Hello";
char str2[] = "Yogesh";
char str3[] = Strcat(str1, str2);
cout << str3;
// output => Hello Yogesh
```

3) Strcpy(): Copy one string to another

```
char str1[] = "C++";
char str2[] = "Python";
strcpy(str2, str1);
cout << "Str2" << str2;
```

output

// Str2 = C++