```
Encapsulation obj;
obj.set(5);
cout<<obj.get()<<endl;
return 0;
}
```

## Friend function and Friend class

If a class become a friend class to other class then it can excess all the func., that is either private or protected member of a class.

Friend class is declared always with the friend keyword.

Syntax

```
class A
{
Public:
Member func()
{
}
friend class B;
};
```

```
class B
{
Public:
Member func(A obj)
{
}
};
```

eg
```
#include <iostream>
#include <conio>

using namespace std;

class A
{
int a, b;
Public:
void input ()
{
cout << "Enter the no:";
cin >> a >> b;
}
friend class B;
};
class B
{
int c;
Public:
void add (A obj)
{
c = obj a + Obj b;
cout << "Sum = " << c;
}
};
void main ()
{
clrscr ();
A oo;
```

```
B KK;
OO input();
KK.add(OO);
getch();
}
```

Friend Function - It is a func° which is not the member of the class instead of that it can access private and protected member of class.
Friend func° is declared as in the class with friend keyword.
Friend func° can become friend to more than one class.

Syntax

```
friend return_type fun_name (class_name &ject)
{
  Body
}
```

eg-

```
void add (A obj)
{
int c;
c = obj.a + obj.b;
cout << "Sum= " << c;
}
int main ()
{
A KK;
KK input ();
add (KK);
return 0;
}
```

What is the basic difference btw friend func. and friend class?

PAGE NO. :
DATE : / /

(more than one friend)

Syntax for nesting of friend class :

```
class A                    class B                    class C
{                          {                          {
int a;                     int b;                     int c
Public:                    Public:                    Public:
 void putdata();           void getdata();            void data();
 friend class B;           friend class D;            };
 friend class C;           };
};
```

Constructor- (for creating)

Constructor are the special member fun. of a class which is used to initialise values of the variable inside an object.

major pts about constructor-

1. Constructor's name is always same as the class name.
2. Constructor is automatically invoked as soon as an object of its class is created.
3. A constructor hasn't any ~~return type~~ return type not even void
4. Constructor allows default argument concept.
5. Constructor can neither be inherited nor virtual.
6. Constructor are defined inside the public section.

PAGE NO.:
DATE: / /

Types of Constructor

Custom constructor are classified in three categories -

Default Constructor
Parameterized Constructor
Copy Constructor

Syntax

```
Class Classname
{
Private;
  ≡
Public:
Classname() // Constructor created
 {
  ≡
 };
```

Dummy eg →

```
class Student
{
Public:
Student()
{
cout << " constructor created ";
}
};
int main()
```

```
{
Student S1;
return 0;
}
```

eg→

```
#include <iostream>
#include <conio>
using namespace std;
class constructor
{
Public:
constructor ()
{cout << "constructor created";
}
~constructor()
{cout <<"bye";}
};
void main ()
{
Constructor const;
}
```

constructor created
bye

## Parameterized Constructor

A constructor is a special member function whose name is same as class name and which is automatically invoked as soon as object of its class created.

A constructor having argument is called Parameterized Constructor

PAGE NO.:
DATE: / /

Syntax

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

Syntax / dummy code of parameterized constructor

eg→

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

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

definition of demo constructor

```
{
cout << "a= " << a <<
}
};
int main ()
{
int x, y;
cout << " Enter 2 no. ";
cin >> x >> y;
dumo a a(x, y);
a.a Putdata ();
getch ();
return 0;
};
```

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

When we used to initialize the variable of an ~~to~~ obj with the values of another obj of same ~~type~~ time then we use the concept of <u>copy constructor</u>

<u>Syntax / dummy code</u>

```
class dumo
dumo a.a;
dumo bb = aa ; or dumo bb(aa);
```

PAGE NO.:
DATE: / /

eg→
```
# include <iostream>
# include <conio>
using namespace std;
class demo
{
int a;
Public:
demo () //default constructor
{
a=10;
}
demo (demo &z) //copy constructor
{
a=z.a;
}
void Putdata ()
{
cout <<"\n a= "<<a;
}
};
int main ()
{
demo aa;
demo bb(aa);
demo cc=bb;
aa.Putdata ();
bb.Putdata ();
cc.Putdata ();
getch ();
```

a=10
a=10
a=10

return 0;
}

## Destructor in C++

### difference btw constructor & destructor

| Constructor | Destructor |
|---|---|
| Constructor is used to construct values | Destructor is used to destroy objects when it is not used or end of the scope of object |
| Constructor has an argument but it didn't have any ~~return~~ type. | Destructor neither have an argument nor return type |
| Constructor's name is similar to the class name | Its name is also similar to class name but before class name provided by ~~tilted~~ (~) the tilted sign. |

```
class ABC
{
ABC ()
{ = }
};
```

```
class ABC
{ ABC()
  { = }
  ~ABC()
  { = }
};
```

PAGE NO.:
DATE: / /

Like constructors, destructor are also member fun. of class whose name is similar to the class name having tilled (~) sign just before its name.
A destructor is used to destroy an object, once the object goes out of scope. A destructor has no return type as well as it doesn't take any argument.

Syntax / Dummy code.

```
class ABC
{
 ABC()
 {
  =
  =
  }
~ABC()
{
 =
 =
 }
 };
```

Example-

```
#include <iostream.h>
int count = 0;
class demo
{
Public :
```

count ++ = count+1

0+1

no. of obj created ⇒ 1

PAGE NO.:

DATE: / /

```
demo()
 {
  count ++;
cout <<"\n no. of objut vuated"<< count;
 }
 ~ demo()
 {
 count --;
 cout <<"\n no. of objut deluted"<< count;
  }
 };
 int main()
 {
 demo aa, bb, cc;
  {
  demo dd;
 }
 return 0;
 }
```

| |
|---|
| No. of obj created 1 |
| " " " " 2 |
| " " " 3 |
| " " " 4 |

| |
|---|
| No. of obj deleted 3 |
| " " " " 2 |
| " " " " 1 |
| " " " " 0 |

## Function Overloading in C++

C++ support the concept of polymorphism (more than one forms)

Types of Polymorphism →

1. Function overloading
2. Operator overloading
3. Constructor overloading

## definition

Function overloading concept of polymorphism enables us to create same name multiple func (methods) within a program.
But we have a restriction that all the func. to having same name must follow differences should be in

1) No of argument
2) types of argument

void add (int, int); ✓
void add (int, int, int); ✓
void add (int, int);
void add (int, float);

eg of func. overloading :-

| S.No | Function | No. of argument | Type of argument | |
|---|---|---|---|---|
| 1. | 1 void add (int, int); | 2 | int, int | Not follow func. overloading |
| | void add (int, int); | 2 | int, int | |
| | 2 void add (int, int); | 2 | int, int | followed |
| | void add (int, int, int); | 3 | int, int, int | |
| | void add (int, float); | 2 | int, float | |

PAGE NO. :
DATE : / /

eg→ Prog①

```
#include <iostream>
#include <conio>
using namespace std;
void add( int a, int b)
{
cout << a+b <<"\n";
}
void add (int a, int b, int c)
{
cout << a+b+c<<"\n";
}
int main()
{
add(2,3);
add(5,10,5);
getch();
return 0;
}
```

| output |
|---|
| 5 |
| 30 |

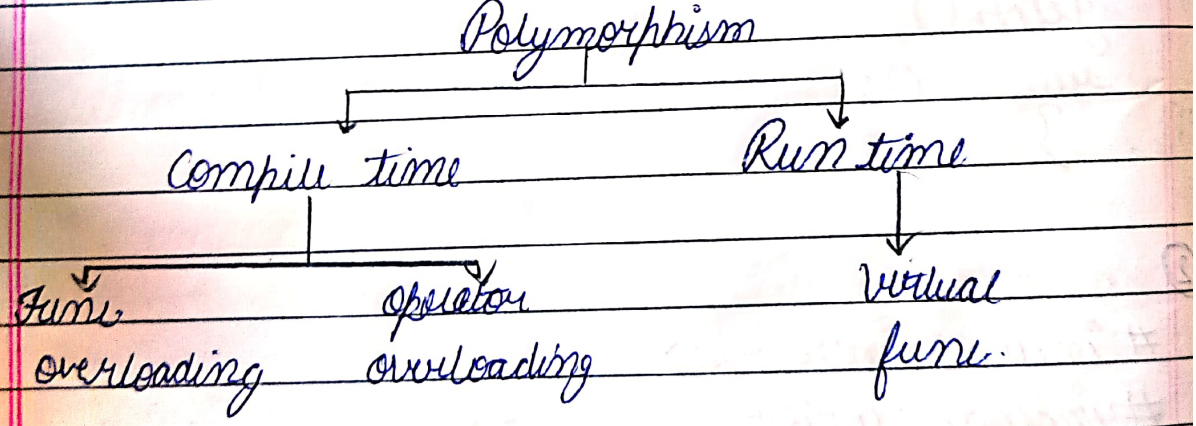
Prog-②

```
#include <iostream>
#include <conio>
using namespace std;
void add (int, int);
void add ( int, int, int);
void add (int, float);
void main ()
{
add (2, 3, 7);
add (5, 2);
```

```
add (7, 1.0);
getch ();
}
void add (int a, int b)
{
cout << "\n add= " << a+b;
}
void add (int a, int b, int c)
{
cout << "\n add = " << a+b+c;
}
```

Output
add = 12
add = 7
add = 8.0

## Operator Overloading




Operator overloading is one of the most imp. feature of C++ that is a type of polymorphism

Using the concept of operator overloading, we can overload any built-in operator that is, we can assign a new definition to an existing

PAGE NO.:
DATE: / /

Operator.

Some operators can't be overload.
Scope resolution operator (::)
Class member access operator (.,*)
Size of operator (Sizeof)
Conditional operator (?:)

The process of overloading involves the following steps-

Create a class that define the data type that is to be used in the overloading operation.
Declare the operator function operator+() or operator op() in the public part of the class. It may be either a member func. or a friend func.
Define the operator function to implement the required operation.

Operator overloading

Unary operator overloading
binary operator overloading

Unary → Syntax
OPX or XOP
Binary → Syntax
XOPY

PAGE NO.:
DATE: / /

# Binary Operator Overloading -

```
#include<iostream.h>
 class demo
{
 int a;
 Public:
 void getdata()
{
cout<<"\n Enter a no.";
 cin>>a;
}
 void Putdata()
{
cout<<"\n value="<<a;
}
demo operator+(demo bb)
{                          pass as an argument
 demo cc;
 cc. a=a+bb.a;
 return cc;
 }
 };
int main()
{
 demo aa, bb, cc;
 aa.getdata();
 bb.getdata();
  cc= aa+bb;
aa.Putdata();
```

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

Unary Operator Overloading

```
# include <iostream.h>
  class demo
{
  int x, y, z;
  Public:
void getdata (int a, int b, int c);
void display (void);
void operator - ();
  };
void demo :: getdata (int a, int b, int c);
  {
  x = a;
  y = b;
  z = c;
  }
  void demo :: display - (void)
{
  cout << "x = " << x "\n";
  cout << "y = " << y "\n";
  cout << "z = " << z "\n";
}
  void demo :: operator-()
{
```

```
    x = -x;
    y = -y;
    z = -z;
}
int main()
{
    demo dd;
    dd.getdata(10, -20, 30);
    cout << "dd";
    dd.display();
    -dd;
    cout << "-dd:";
    dd.display();
    return 0;
}
```

output:

```
dd:
x=10
y=-20
z=30
-dd:
x=-10
y=+20
z=30
```

## Inheritance

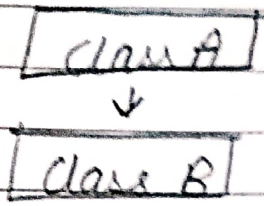
Inheritance is the process by which the class can aquires the properties and method of another class. The mechanism of deriving a new class (derived class) from an old class (existing class / parent class) is called inheritance.

child class

The new class is also called derived class and the old class is also called base class.

The derived class may have all the features of base class and also the programmer can add to the derived class

General Block diagram of Inheritance




Here the class A is called the base class / Old class / Parent class / existing class because this class alrea exist and the features of class A is inherited by class B so class B is called derived class / new class child class. Here the class B having features of cl and also the programmer can add some new features.

Defining derived class:-
A derived class can be defined by specifying its relationship with the base class in addition to its o details. The general form of defining a derived class.

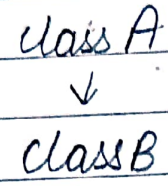
Syntax: class derived_class_name: visibility_mode base_class_na

eg→ class B :: Protected A
{
// member of derived class
}

Types of Inheritance
1 Single level
2 Multi level
3 Multiple
4 Hybrid
5 Hierarchial

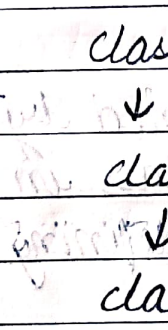
PAGE NO.:
DATE: / /

1. Single level Inheritance -




If a class is derived from a single class then this type of inheritance is called single level. In the above block dig. class B is derived from class A.

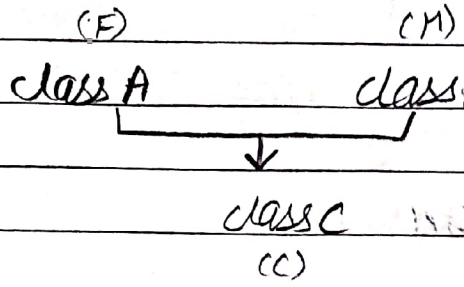
2. Multi level Inheritance -




If a class is derived from a class which is derived from another class then it is called multi level inheritance.
Here the class C is derived from class B and class B is derived from class A. It means class B is child of class A & it is parent of class C, so it is called multilevel inheritance.

PAGE NO. :
DATE : / /

3. Multiple Inheritance



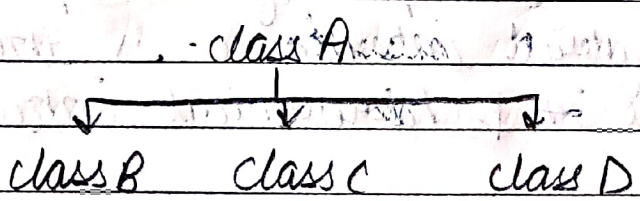

In this type of inheritance, a class is derived from more than one class.

OR

If a class is derived from two or more than two classes then it is called multiple inheritance

In above dig. class C is derived from two classes, class A and class B

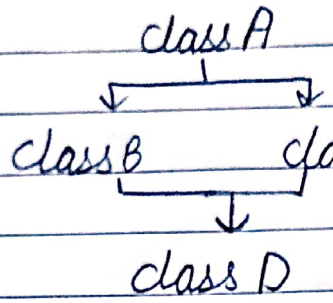
4. Hierarchical Inheritance




If one or more classes are derived from one class then it is called hierarchical inheritances.

Here the class B, class C & class D derived from single class that is, class A.

5. Hybrid class




Hybrid inheritance is the combination of any above inheritance type i.e either multiple or multilevel or hierarchical or any other combination.
Here, class B and class C are derived from class A, and class D is derived from class B and class C.
Class A, class B and class C is an example of hierarchical inheritance & class B, class C and class D is an example of multiple inheritance.
So the hybrid inheritance is the combination of Multiple & hierarchical inheritance.

Constructor Overloading
C++ provides us the provision using which we can incorporate more than one constructor in a single prog But these constructor must have diff types of arguments / diff no. of arguments.

This provision of having more than one constructor in a single program is called constructor overloading

PAGE NO.:
DATE: / /

Q→

```
#include<iostream>
#include<conio>
using namespace std;
class demo
{
int a;
Public:
   demo() //default
   {
    a=10;
   }
   demo(int x) //Paramterized
   {
   a=x;
   }
   demo (&z) //copy
   {
   a=z.a;
   }
   void Putdata()
   {
   cout"\n A= "<<a;
   }
   };
intmain()
{
demo aa;
demo bb(aa);
demo cc(bb);
return 0;
}
```

Output

A = 10
A = 20
A = 10

PAGE NO.:
DATE: / /

Inheritance example-

eg → (Single level)

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

class base
{
Public:
int x;
void getdata()
{
cout<<"Enter the value of x";
cin>>x;
}
};
class derived: Public base
{
Private:
 int y;
Public:
 void readdata()
{
 cout<<"Enter the value of y";
 cin>>y;
}
void Product()
{
cout<<"Product= "<<x*y;
```

OUTPUT

```
Enter the value of x
5
Enter the value of y
4
Product
20
```

```
};
int main()
derive aa;
aa.getdata();
aa.readdata();
aa.product();
return 0;
}
```

up⇒ (Multi level)

```
#include<iostream>
using namespace std;
class base
{
public:
int x;
void getdata()
{
cout<<" Enter the value of x";
cin>>x;
}
class derive: Public base
{
Private:
int y;
Public:
void readdata()
{
```

```
cout <<" Enter the value of y";
cin >>y;
 }
};
class child : Public derived
 {
void product ()
{
```

PAGE NO.:
DATE: / /

## Visibility of Inherited Members

| Base Class | Derived Class | | |
|---|---|---|---|
| | Public | Private | Protected |
| Public | Not Inherited | Not Inherited | Not Inherited |
| Private | Protected | Private | Protected |
| Protected | Public | Private | Protected |

→ eg (Multiple Inheritance)

```
Dummy → classbase1
        {
        ==
        };
        classbase2
        {
        ==
        };
        class child : Public base1, Public base2
        {
        ==
        }
```

Syntax-

```
class derived : Public b1, Public b2, --- Public bn
{
};
```

PAGE NO. :
DATE : / /

eg Hierarchical Inheritance

Dummy →

```
class base
{
==      (x,y)
};
class c1 : Public base
{
==      Product ()
        (x*y);
};
class c2 : Public base
{
==      Add
        Return (x+y);
};
```

eg →

PAGE NO. :
DATE : / /

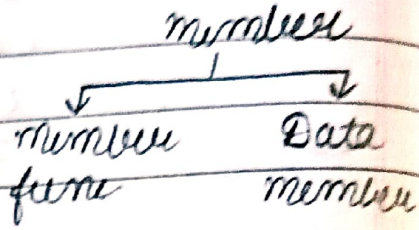
eg Hybrid Inheritance

```
Dummy: class base
       {
       ≡
       };
       class c1 : Public base
       {
       ≡
       };
       class c2 : Public base
       {
       ≡
       };
```

method overriding?

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

output
Derived class




The above program shows the concept of Method overriding because here derived class method overrides the base class method.

Pointer is always created for base class.
Pointer (Ptr) is always associated with ~~by~~ ~~binary / runtime~~

PAGE NO.:
DATE: / /

```
#include<iostream.h>
class A
{
public:
  void show()
  {
  cout<<"\n Base Class";
  }
};
class B : public class A
  {
  Public:
  void show ()
  {
  cout<<"\n Derived class";
  }
  };
  int main()
  {
  A* bPtr;
  A aa;
  bPtr = &aa;
  bPtr -> show
  return 0;
  }
```

Output
Base class

class A pointer created

A aa; → Obj of class A created

bPtr = &aa; → Pointer is associated with object

→ - sign and greater than

pointer calls the show function

example of virtual Function

```
#include <iostream.h>
class A
{
public:
virtual void show()
```

PAGE NO. :
DATE : / /

Virtual is a keyword in C++.
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.

Pure virtual function
A virtual function having no definition is called pure virtual function.

eg→

virtual void show() = 0

DATE: / /

Static & constant members

- Data Member
- Member Function

Static member
- Static Data member
- Static member Function

constant member
- const data member
- const member function

Static data member is initialize to zero whenever the first object of its class is created no other initialization is permitted. for making any data member static, we use static keyword Only one copy of static data member is created and shared by all its visibility is in entire program

z value initialize from 0

PAGE NO.:
DATE: / /

dummy code

x

| aa | a | bb | a |
|---|---|---|---|
| | b | | b |
| | x | | x |
| | getdata | | getdata |
| | setdata | | setdata |

```
class demo
{
int a,b;
  int x;
Public:
  void getdata ();
  void setdata ();
};
int main ()
{
demo aa, bb;
```

example-

```
#include <iostream.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 = "<< "\t z = "<<z;
```

Formal

x=5 y=7 z=0
x=20 y=10 z=1

Static data member depend on static data function

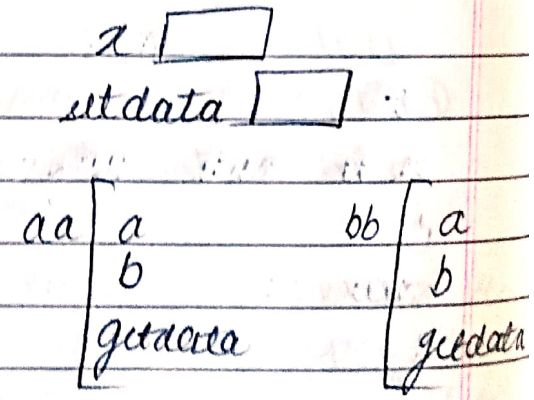
```
}
};
Int demo::z;
int main ()
{
demo aa, bb;
aa.getdata (5,7);
bb.getdata (20,10);
aa.putdata ();
bb.putdata ();
return 0;
}
```

## Static Member Function

Static Member Function can access only static data member.

It is not part of any object, it is call using to class name

dummy code




```
class demo
{
int a,b;
Static int x;
Public:
 void getdata ();
Static void setdata ();
 };
int demo :: setdata ();
```

PAGE NO.:
DATE: / /

```
int main()
{
 demo aa, bb;
```

example-

```
#include<iostream.h>
  class demo
  {
  int x;
  static int y;
 Public:
 void getdata (int a)
  {
   x=0;
   y=y+1;
  }
   void putdata ()
   {
   cout<<"\n x= "<<x<<"\t y= "<<y;
   }
   Static void ab();
   {
   cout<<"\n y= "<<y;
   }
   };
   int demo::y;
   int main()
   {
   demo aa;
```