Coaching

Date
Page

C++

① introduction ② Structure of C++ program
③ character set of C++ ④ Token of C++

Tokens of C++
- keywords — 63 bytes
- identifier
- constant
- Operator
- seperator

Output statement :- (Cout) → c out

① The output statement is use to print the message / statement, value of the variable & result of expression to the console

② It works similar as printf in C

Syntax :- i) Print Message :-

cout << " MESSAGE " ;

printf("Message");
↳ f" (pre-defined
↳ stdio.h

⇒ Cout :- (object) It is a predefine object of Ostream class.

⇒ << :- It is a insertion operator or put to operator

cout << " Hello " ;

→ print as it is

→ Hello

Here Cout represents monitor or computer screen

ii) print value of variable :-

Syntax:

cout << variable name ;

eg cout<< a ;

Scanned by CamScanner

```
int a=10;
cout << a;
cout << 10;
cout << t ;  // error
```

Cascading of cout (insertion operator) :-

```
cout << var1 << var2 << ---- << var n ;
```

eg.
```
int a=5, b=10, c=3;
cout << a ;  cout << b ;  cout << c ;
```

(or) `cout << a << b << c ;`	`cout << abc ;` ✗ `cout << a, b, c ;` ✗

eg.
```
int a=5;
cout << "value of a=" << a ;
```

eg.
```
int a=5, b=10;
cout << "value of a=" << a << " & b=" << b ;
```

Input Statement :- (Cin)

(i) Input statement is used to scan or read the value from console entered by user.

(ii) It works similar as "scanf" in C.

Syntax :- `cin >> var ;`

eg.
```
int a;
cin >> a;
```

cin :- The cin is a predefine object of class istream

>> :- It is a extraction operator or get from operator

cout <<
cin >>

Cascading of extraction operator :-

cin >> var1 >> var2 >> --- >> var n ;

eg
```
int a, b, c ;
scanf ("%d%d%d", &a, &b, &c);
```
/c

```
int a, b, c ;
cin >> a ;
cin >> b ;
cin >> c ;
```

```
int a, b, c ;
cin >> a >> b >> c ;
```

prog P1
```
void main ()
{
    int a, b, c ;
    clrscr ;
    cin >> a >> b ;
    c = a+b ;
    cout << " sum = " << c ;
```
header file ⇒ (iomanio.h or manio.h)

Manuplator :- ① endl ② setw

1) **endl :-** → It stands for end line.
→ It works similar as new line character \n.

eg. int a=5, b=10 ;
~~cout << a << b~~
```
cout << " value of a = " << a << endl ;
cout << " value of b = " << b ;
```

② **Setw :-** ⇒ (set width)
→ It is used to set the width of no. print on console.

Scanned by CamScanner

eg. int a= 73;

cout << a << endl << setw(4) << a << endl << setw(1) << a ;

```
73
   73
73
```

Control Statements :- i) decision making statements
↳ a) if
b) if-else
c) if-else ~~nested~~ ladder
d) nested if else
e) Switch - case

ii) looping / Repetative Statements :- a) while b) do-while
c) for

iii) Jump statements :- 1) go to 2) break 3) continue

Reference variable (not in ( ) :- It is used to create alias (उपनाम) for alternative name of the already existing variable.

→ It doesn't create a separate Memory

Syntax :- data-type &new-name = old-name
OR

data-type &reference-name = old-name
↑
amperoand

1/Aug/2019

C++

eg.

```
int a=5;
b=a;
```

a: 5 (100), b: 5 (102)

```
int a=5;
int &b = a;
```

a/b: 5 (100)

eg.

```
void Main ()
{
int m=10;
int &n=m;
~~void main ()~~ n++;
cout<< "m=" <<m << endl;
~~cout <~~ n= m*10;
cout << "n=" << n << endl;
}
```

m: 11 (110)

eg `int &n = 50;` // valid

int & → Reference to int

float & → Reference to float

Bool :- (new data type in C++) It is used to store the value of boolean type

boolean → 0/1

0 → false, 1 → True

⇒ Size of bool is 1 byte

eg.

```
bool b1 = true;
bool b2 = false;
```

(true: keyword)

```
bool b1 = 0;
bool b2 = 1;
```

eg.

```
void Main ()
{
bool b1=true, b2=1, b3;
int c;
c= b1+b2
b3 = b1+b2
```

c = b1+b2 = 1+1 = 2

b3 = b1+b2 = 1+1 = 2 = true

bool type = 1

```
cout << "c=" << c << endl;
cout << "b3=" << b3;
}
```

2
1

non-zero = True
zero = false

eg.

```
void main ()
{
    int a=5, b=2;
    cout << 25 << endl;
    cout << 2.4 << endl;
    cout << 'p' << endl;
    cout << "xy" << endl;
    cout << -5/2 << endl;
    cout << a%(-b) << endl;
```

↳ Remainder

15
2.4
p
xy
-2
1

Function :-

→ The f^n is a self contain body which is used to perform particular task.

→ Type of f^n :- i) pre-defined (in-built) f^n ← printf, cin, getch, scanf, strlen
ii) user defined

user-defined f^n :- we can also create own f^n to perform a particular task

→ ~~Necessary~~

Scanned by CamScanner

Neccessary Statement for OOP :-

i) f^n declaration / prototype
ii) f^n call
iii) f^n defination
iv) return statement

f^n declaration :-

Syntax :- ~~f^n name~~

return_type f^n name (list of arguments with individual data-type) ;

formal arguments / parameter (inputs)

↳ seperated by comma

↓ datatype of return value

(or)

return type f^n name (list of datatype w/o argument name) ;

eg $\boxed{\text{int add (int a, int b);}}$

or

$\boxed{\text{int add (int ; int);}}$

f^n defination :-

syntax :

return-type f^n name (list of arguments with individual datatype)

formal argument / parameter

{
//body
return statement ;
},

eg $\boxed{\text{int add (int, int) ;}}$

```
int add (int x, int y)
{
int z ;
  z = x + y ;
  return (z);
}
```

Scanned by CamScanner

f^n call :- syntax :-

f^n-name (list of argument/parameter without datatype); (actual)

eg.

```
add (a, b);
   or
add (5, 2);
```

```
cout << a+b;
   ⇓
cout << add(a,b);
```

```
C = a+b;
   ⇓
c = add(a,b);
```

Return statement :- It is used to return a value from the f^n to the calling statement

Syntax :-

i) return (var/ expression / value) ;

ii) return & var/ expression/ value ;

iii) return

eg

```
return (5);   ✓
return (5, 2);   ✗
return (5+2);   ✓
return (a+b*c);   ✓
return (a);   ✓
return a+b;   ✓
```

→ A f^n return only one value at a time.

→ The ~~return~~ return statement may be ~~written~~ written many times in a f^n but works only one at a time.

Scanned by CamScanner

Q1 WAP to find sum of 2 integer 7^n using f^n.

Q2 WAP to find greatest no. among three different no.

Q. WAP to check given no. is prime or not using f^n.

Pointers: i) pointer is a variable which holds the address of other existing variable.

ii) Through the pointer we can access the variable which being pointed by that pointer.

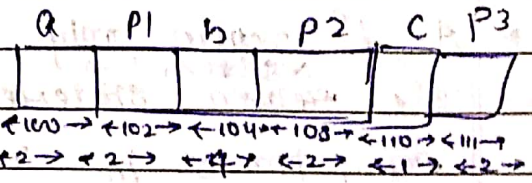
declaration of pointer :-

$\boxed{\text{data-type} \quad * \text{ptr-name} \; ;}$

Astrick

ex. int *P1 ;
float *P2 ;

ex. int a, *P1 ;
float b, *P2 ;
char c, *P3 ;



pointer variable always allocate with 2 bytes compiler dependend = (- size of (int)) because it hold the address and address is of integer type int *P1 , float *P2 ; char *P3 ;

→ data-type *ptr ;

C++

Date
Page

Here the datatype is a type of variable which being pointed by ptr.

ex. int *P1 → Here P1 is a pointer variable which can point the variable of only & only integer type.

ex. int a=5 , *p ;
p= &a

a	p
5	100
100	102

cout << a << endl ;
cout << p << endl ;
cout << &a << endl ;
cout << *p << endl ;

5
100
100
5

*p

Indirection operator/ deRefrencing operator

*p = value of address hold by p
or
value of variable pointed by p

[&a or p] . [a or *p]

ex. void main ()
{
int a=5, b=10 , *p1 , *p2 ;
p1 = &a ;
p2 = &b ;
a++ ;
cout << a << " is stored at " << p1 << endl ;
cout << b << " is stored at " << &b << endl ;
cout << *p1 << " is stored at " << &a << endl ;

Scanned by CamScanner

```
cout << b << " is stored at " << p2 << endl;
cout << a << " is stored at " << &p1 << endl;
cout << *&b << " is stored at " << p2 << endl;
        *p1 = *p1 + 5;
cout << " a= " << a << " and b= " << b;
```

a	b	p1	p2
5	10	100	102
100	102	104	106

6 is stored at 100
10 ——— 102
6 ——— 100
10 ——— 102
6 ——— 100
10 ——— 102
a = 11 & b = 10

f^n calling :- In C++ a f^n may called through different ways

(1) call by value (2) call by address (3) call by Reference

call by value :- In this mechanism the value of arguments are passed to the calling f^n.

call by address :- In this approach the address of arguments are passed to the calling f^n. In this mechanism we must use pointers to hold the address of passed arguments

call by reference :- In this mechanism the references are crete for passing arguments in the calling f^n.

Scanned by CamScanner

C++

f^n with default argument :- In C++ we can create a f^n with default values for the arguments.

ii) The default values or the arg. is set at the time of f^n declaration.

iii) If we set some arg. with default value then we call the f^n w/o them.

ex. void simple-Interest (int p, int t, float rate) ; default value
or
void simple-Interest (int p, int t, float rate = 12.5) ;

calling :-

a) simple-Interest(5000, 2, 13.5); ⇒ p = 5000, t = 2, rate = 13.5

b) simple interest (5000, 2) ; p = 5000, t = 2, rate = 12.5

c) simple-Interest (5000,) // Invalid
↳ we must pass value for non-default parameter / arg

ex. void simple-Interest (int p, int t = 4, float rate = 12.5);

Call :- simple-Interest () ; // Invalid
simple-Interest (5000) ; // valid
simple-Interest (5000, 6) ; // valid
simple-Interest (5000, 6, 13.5) ; // valid

we the default arg. must be declare from right to left bcx the value of arg./parameter are copied corresponding

Ex. void Simple_Interest (int P , int t = 4, float rate); // invalid

Call :- 1. simple_Interest (5000, 12.5) ; // Invalid

2. Simple_Interest (5000, 4, 12.5); ⇒ We must passed three parameter

Ex.

```
void print ( int n=10 , char ch= '@' );
void main ()
{
    print () ;
    print (20) ;
    print (20, '$' );
}

void print ( int n , char ch )
{
    int i ;
    for ( i= 1 ; i<= n ; i++ )
    {
        cout << ch << " " ;
    }
    cout << endl ;
}
```

@ @ @ @ ---- 10 times
@ ------- 20 times
$ $ $ --- 20 times

Memory allocation f^n :- In C++ we can allocate the memory in two ways i) static memory allocation
ii) dynamic memory allocation.

Static M.A :- This type of allocation is done at the compile time (compilation)

ii) we can't change static memory allocation if it allocated ones.

iii) for this type of allocation we doesn't need any extra predefined f^n

Ex i) int a, b, c ;

ii) int a[10];

iii) struct xyz
{ int m;
int n;
} p ;
↳ variable

2) dynamic memory allocation :- This type of memory allocation is done at the time of run (execution time).

ii) we change or modify dynamic allocated memory wherever required.

iii) for this allocation we use the special pre-defined f^n/operator

~~Memory~~ :- Memory allocation f^n

In C	In C++
1) Malloc ()	1) new → Memory allocation (creation)
2) Calloc ()	2) delete → Memory deallocation (destroy)
3) free ()	
4) realloc ()	

⇒ In C the malloc(), calloc(), free() & realloc() are the pre-defined f^n but in the C++ 'new' & 'delete' are the pre-defined operator.

i) malloc():- Syntax:-

```
datatype *ptr;
ptr = (castype *) malloc (size);
```

byte

ex int *p;

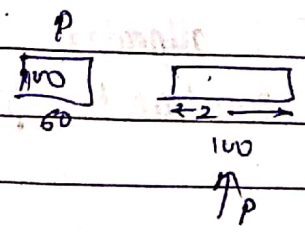
p = (int *) malloc (2);

↳ casting 2 bytes

⇓

*p = 10;

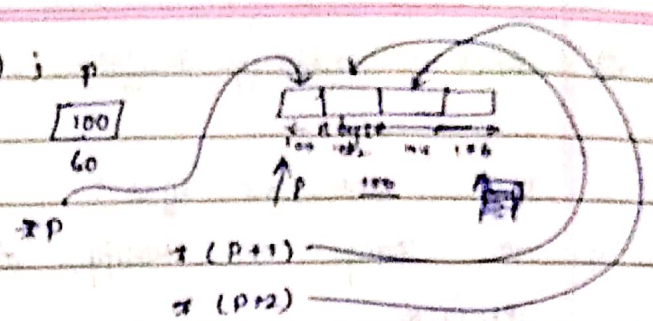
printf("%d", *p);



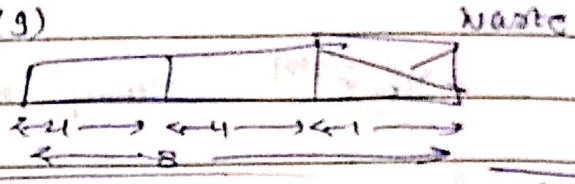
scanf ("%d", &n);

int *p = (int *) malloc (n * sizeof (int));

Ex. int *p = (int *) malloc (8); p



ex float *p = (float *) malloc (9)



float *p = (float *) malloc (2 * sizeof (float));

2) Calloc :- It is used to create multiple blocks of given size

Syntax :-

data-type *ptr;

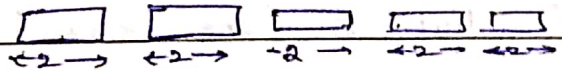
ptr = (cast-type *) calloc (n, size);

no. of block

↳ size of each block

Ex. int *p;

p = (int *) calloc (5, sizeof (int));



3) free () :- It is used to deallocate or destroy memory dynamically.

Syntax :- free (ptr);

4) realloc () :- It is used to modify the size of memory allocated dynamically.

realloc (ptr, new-size);

Scanned by CamScanner

1) **New operator :-** → It is used to allocate memory dynamically
→ It works similar as malloc() in the C.

Syntax :

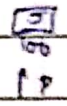
data-type * ptr = new_datatype ;

Ex

int *p = new int ;
float *q = new float ;
char *r = new char ;

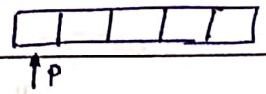
Initialization :-

int *p = new int (5) ;
↑ Initialization



float *q = new float (12.5) ;
char *r = new char ('t') ;

Array : int *p = new int [5] ;
↓
array of 5 elements of int type



2) **delete Operator :-** It is used to destroy or free or release dynamic memory allocated by new operator.

Syntax :- delete ptr ;
array → delete [] ptr ;

ex int *p = new int [10];

In this statement if the memory is not available or the memory is not created then the null is assign to p if the memory is create then it assign the starting address to the p.

Scanned by CamScanner

```
if (2P)
{
  cout << "memory is not available\n";
}
```

f^n overloading :-

Overloading (Polymorphism) :- overloading means use of same thing for the different purpose

Poly - many , morphism - form

polymorphism is a concept in which one thing shows the different behaviour at different stand.-

- polymorphism
  - static
    - f^n overloading
    - operator overloading
      - eg $5+2=7$ (arithmetic addition)
      - "Hi" + "BYE" ⇒ "Hi BYE" (concate the string)
  - dynamic
    - f^n overriding (Through virtual f^n).

→ The f^n overloading is the imp. features of the OOPS (C++)

→ In C++ we can create a many f^n of same name but with the diff. arg. list to perform the diff task.

Scanned by CamScanner

C++

Ex. int add (int, int, int) ; — I
int add (int, int) ; — II
float add (float, float) ; — III

Calling Statement :- add (5,4) ; → call - II
add (5, 4, 3) ; → call - I
add (2.5, 4.0) ; → call - III

Steps to resolve the selection of overloaded f^n :- (taken by compiler)

(1) First of all the compiler check the no. of arg. in the calling f^n. Then it tries to find exact datatype match for the arguments

(2) If the step 1 fails (If the exact match are not found) then it tries to convert integral conversion.
Char → int
float → double

(3) If compiler fails then it tries to convert implicit conversion.
int → float
int → long
~~float~~ → double

If there is a possibility to convert in many conversion like

$\boxed{\text{int} \rightarrow \text{float}, \text{int} \rightarrow \text{long}}$

void xyz (float) ;
void xyz (long) ;

call.
xyz (5) →

Scanned by CamScanner

Here the int may be convert either float or long so the Compiler get confused. This type of conversion produces an error.

ex.
```
long volume ( int l, int b, int h)
    { return (l*b*h); }
float volume (int r, int h)
    { return ( 3.14 * r * r * h ); }
int volume (int a)
    { return (a*a*a); }
void main ()
{ cout << " volume of cylinder : \n" << volume (5,4) << endl;
  cout << " volume of cube : \n" << volume (5) << endl;
  cout << " volume of cuboid : \n" << volume (3,4,5) << endl;
}
```

Returning Reference :- In C++ we can implement a f^n with returning references

ii) Till now we put the calling statement always at the right hand side of the assignment operator but in the C++ we can write the calling statement at L.H.S of assignment operator but we must use returning with reference concept.

call by reference

ampersand

ex
```
int & greater (int &x, int &y)
    { if (x>y)
        return (x);
      else
        return (y);
    }
```

Scanned by CamScanner

```
void main ()
{
    int a=15 , b=10;
    cout << " a= " << a << " & b = " << b << endl;
    cout << " the greter no. is reward 501 \n" ;
    greter (a, b) = 501 ;
    cout << " After reward \n" ;
    cout << " a= " << a << " b= " << b ;
}
```

Inline f^n :- • It is a special feature in C++

- Inline f^n is used to reduce or eliminate the cost in memory or time when calling f^n.
- It is rarely used where for the small code or small f^n.

- for making the f^n inline we just use keyword "inline". In front of f^n header

```
inline return_type f^n name ( arg. list)
{
    // body
}
```

The inline f^n are expand in line where from the f^n called it work similar as 'Macro' (# define)

ex. inline int sum (int x, int y)
{ return (x+y); }

Scanned by CamScanner

```
inline int Sub (int x, int y)
    {
        return (x-y);
    }
void main ()
{
    int a = 15, b = 10;
    cout << " sum of " << a << " & " << b << ":" << sum (a, b) << endl;
    cout << " Sub of " << a << " & " << b << ":" << Sub (a, b) << endl;
}
```

```
sum of 15 & 10 : 25
sub of 15 & 10 : 5
```

⇒ The Inline f^n may not work at some places/ body.

i) If f^n have returning statement when it have loop, switch case or goto statement.

ii) if f^n do not returns value but it have return statement.

iii) If f^n contains static variables.

iv) If f^n is recursive.

Scanned by CamScanner

C++

DATE / /
PAGE

Structure :-

```
struct tag_name
{
    datatype    member 1 ;
    datatype    member 2 ;
    :
    datatype    member n ;
};
```

eg.

```
struct XYZ
{
    int m ;
    float n ;
    char k ;
};
```

```
struct student
{
    char sname[10] ;
    int age ;
    char fname[10] ;
};
```

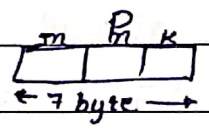
Structure variable declaration :-

```
data_type    variable_name ;
```

```
struct tag_name    variable_name ;
```

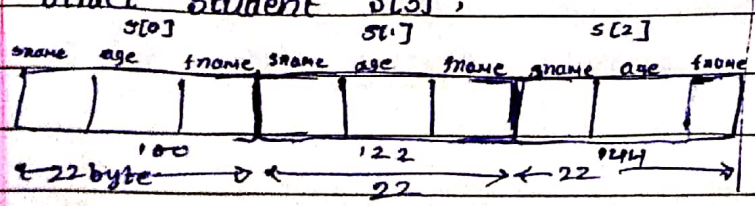
```
struct XYZ    p ;
```

data_type ↳ variable

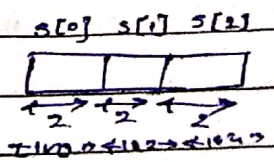



```
int p ;
```

```
struct student    s[3] ;
```

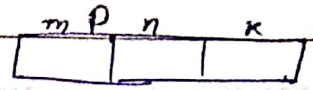



```
int s[3] ;
```



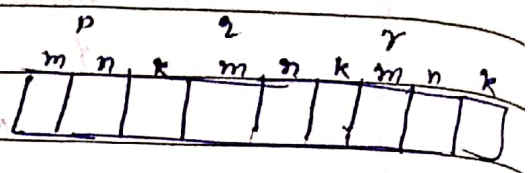
Scanned by CamScanner

Accessing of member :-



p.m = 10

struct xyz p, q, r ;



→ **dot operator :-** $\boxed{\text{Variable-name} \cdot \text{member-name}}$

dot

e.g. p.n = 10 ; p.m = 25 ; p.k = 't'

Class :-

① ⇒ **Data encapsulation :-** It is the features of OOPs in which the wrapping up of data and f^n s together in a single unit. And this unit is called class.

② The class is a user defined data-type like structure & union.

③ The class is a abstract datatype (ADT).

defining the class :-

→ The def. of class is similar to the def. of structure in C. Here the keyword 'class' is used instead of 'struct'.

Scanned by CamScanner

```
Class    class-name
{
  <Access Specifier> datatype    data-member 1 ;
  <  "        "      >   "       data-member 2 ;
        ⋮                                          → data Member
  <  "        "      >   "       data-member n (function) ;

                        return type
  <Access Specifier> ~~datatype~~    member-function 1 () ;
  <  "        "      >                                    → Member fn.
        ⋮
  <  "        "      >   return type   member-function n () ;
};
```

⇒ The data and f^n declare in the class are collectively called member of the class. The data individually called <u>data-member</u> and the f^n are called <u>member function</u>.

<u>Access specifier</u> :- (visibility level)

Here 3 type of access specifier in c++.
i) private ii) public iii) protected

① <u>Private</u> :- a) private member of a class can accessed only through the other member of the class.
It means they can't access outside the class.

b) It is the <u>default</u> access specifier in the c++.

Scanned by CamScanner

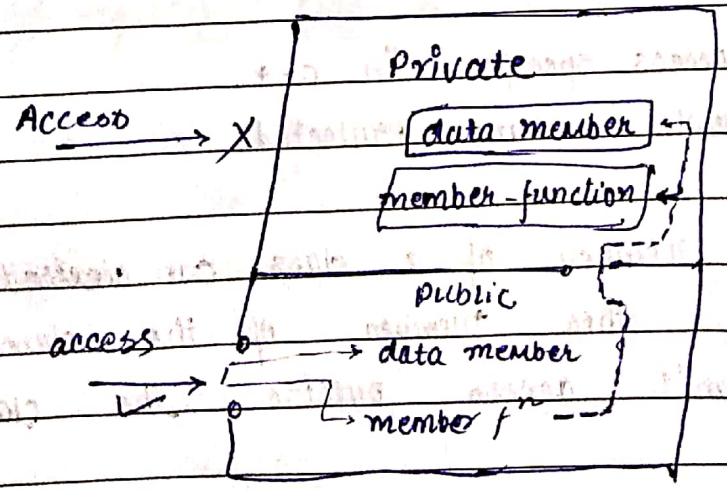
② **Protected** :- The protected member of class can be accessed only through the member of that class and member of subclass.

③ **Public** :- The public member of a class are accessed from any where of program it mean they can accessed through the member f^n and outsider f^n.

eg-

```
class xyz
{
    int m ;
    int n ;        } → data member (both member are of private)
    Public :  void add () ;
              void sub () ;   } → member function
};
```

Data hiding :- It is the concept of OOPS in which we can hide the data from outsider through the keyword 'private' & 'protected'.



Scanned by CamScanner

Object :- Object is a real time entity.

→ When object the class ~~declared~~ define ones then we can create the variable of that class-type & these variables are called object.

→ The class is a blue print for a properties of Object. It means class defines only property only in logical way. not in physical existence.

→ The object is a physical entity it means it have memory allocation in computer memory.

declaration / creation of object :-

Syntax :- `data-type variable name ;`

`class-name object_list ;`

`Struct structure_name variables ;`

In 'C' when we declare the structure variable we must use struct keyword but in C++ we does not use keyword struct and class to defining structure and class.

Ex `XYZ P, Q, R ;`

↑ Class name ↑ objects

Accessing of member of class :- To access the member of class we use dot operator . dot is placed b/w object & member.

Scanned by CamScanner

Access data member :-

```
class XYZ
{
  public :
    int m ;
    int n ;
  Public :
    void add () ;
    void sub (int, int) ;
};
```

Q.

XYZ p, q ;

dot

Objectname . data member

p.m
p.n

Access member function :-

Object name . member function name (Parameter / argument list) ;

eg. p. add () ; p . sub (5, 7) ;
q. add () ; q. sub (7, 8) ;

Q. **Program to add two no. :-**

```
struct XYZ
{
    int m ;
    int n ;
};
void main ()
{
    struct XYZ p ;
    int sum ;
    p.m = 15 ;
    p.n = 10 ;
```

Scanned by CamScanner

```
sum = p.m + p.n;
printf("sum = %d", sum);
}
```

Q. Program to print & input member of structure

```
struct xyz
{
    int m;
    int n;
};
void main()
{
    struct xyz p;
    p.m = 10;
    p.n = 15;
    printf("m = %d & n = %d \n", p.m, p.n);
}
```

→ public

```
class xyz
{
    public : int m;
             int n;
};
void main()
{
    xyz p;
    p.m = 10;
    p.n = 15;
    cout<<" m="<<p.m<<" & n "<<p.n;
}
```

```
class xyz
{
    int m;
    int n;
};
void main()
{
    xyz p;
    p.m = 10; // error
    p.n = 15; // error
    cout<<" m="<<p.m; // error
```

private

```
class xyz
{
    int m; int n;
    public:
    void input()
    {
        m = 10; n = 15;
    }
    void print()
    {
        cout<<" m="<<m<<" & n="<<n;
    }
};
void main()
{
    xyz p;
    p.input();
    p.print();
}
```

Scanned by CamScanner

ex

```
class xyz
{
  int m;
  int n;
Public :  void input (int x, int y)
          {  m=x ;  n=y ;  ---->  cin >> m >> n ;
          }
          void print ()
          { cout <<"m=" << m<<" n=" << n ;
          }
};
void main ()
{
    xyz p, q ;
    p. input (10, 15 );
    cout <<" data of p:" << endl ;
    p. print () ;
    q. input (3,4) ;
    cout << " data of q\n" ;
    q. print ();
}
```

For input from user

void input ()

cout << " enter the value for p : \n" ;

p. input ();

cout << " enter two no. out for q " ;

q. input ();

Q. WAP find the greatest b/w two number.

```
class xyz
{
  int m;
  int n;
Public : void input ( int x, int y)
  {
```

```
    m=x ; n=y ;
}
int greatest ( )
{
    if ( m>n)
        return(m);
    else
        return (n) ;
}};
void main ()
{
    P ;
    int a, b, max ;
    cout <<" enter two no :\n" ;
    cin >> a >> b ;
    P. input (a, b) ;
    max = P. greatest ( );
    cout << " maximum is = " << m ;
}
```

Defining the member function :- The member f^n defination can be define at two places i) inside the class ii) outside the class.

i) **Inside the class :-** we can define the member f^n inside the class in place of member f^n declaration.

b) There is no need to declare member f^n separately

c) The member f^n defined inside the class are treated as inline function.

Scanned by CamScanner

ex

```
class xyz
{
    int m;
    int n;
    Public : void getdata (int x, int y)
                { m=x ; n=y ; }
             void printdata ()
             { cout << " m= " << m << " n= " << n ; }
};
void main ()
{
    xyz p;
    p.getdata (5,2);
    p.printdata ();
}
```

2) Outside the class :- we can define member f^n outside the class.

b) When we want to define member f^n outside the class we use a "membership identity level" which tells the compiler for the class the function belongs to

Syntax :

```
return type  class name :: function name (argument list)
{
    // body
}
```

→ Here we must declare the member function inside the class.

Scanned by CamScanner

ex.
```
class XYZ
{
  int m ;
  int n ;
  Public : void getdata (int , int) ;
           void printdata () ;
};
void XYZ :: getdata (int x, int y)
{
    m = x ; n = y ;
}
void XYZ :: print_data ()
{
    cout << " m = " << m << " & n = " << n ;
}
void main ()
{
    XYZ P ;
    P.getdata (5, 3) ;
    P. printdata () ;
}
```

Nesting of function :- If the one f^n is call the second f^n & second f^n call third f^n and so on , This mechanism is called nesting of f^n.

ex.
```
class XYZ
{
  int m ;
  int n ;
  Public : void getdata (int, int) ;
           void display() ;
           int largest () ;
};
```

Scanned by CamScanner

```
Void xyz :: getdata ( int x, int y)
    {   m=x ; n=y; }
int xyz :: largest ()
    {   if (m>n)
            return m;
        else
            return n ;
    }
Void xyz : display ()
    {
      cout <<" greater no. :" << largest () ;
    }
Void main ()
{
    xyz p ;
    p. getdata (15, 19) ;
    p. display () ;
}
```

Access the private member function :- The private member of the class cannot accessed through the outsider f^n so for accessing private member f^n we must have atleast one another public member function.

Ex

```
class xyz
{
    int m;
    int n;
    Void getdata (int, int);
```

```
Public :  void printdata();
};
void xyz :: getdata (int x, int y)
{     m=x ;  n=y; }
void xyz :: putdata ()
{     getdata (5,3);
      cout<<" m=" << m<<" n=" << n;
}

void main ()
{
    xyz  p ;
   // p. getdata (5,3);
     p. putdata ();
}
```

Array within class :-

→ we can use array as a member of class

```
class student
{
   char sname [20];
   int marks[3];
   Public : void getdata() ;
            void putdata();
};          cout <<"
void student :: getdata()
   {
        cout <<" Enter name : " << endl;
        cin >> sname ;
```

Scanned by CamScanner

```
cout << " Enter marks of three subject :" << endl;
for (int i=0; i<3; i++)
    cin >> marks[i];
}

void student :: putdata()
{
    cout << " Details of student :" << endl;
    cout << " name:" << sname << endl;
    cout << " marks of three subjects : \n" ;
    for (int i=0 ; i<3; i++)
    {
        cout << " sub" << " i+1 " << "=" << marks[i];
    }
}

void main ()
{
    Student s;
    s.getdata();
    s.putdata();
}
```

Array of Objects :- we can declare or create the variable of class as a array which is called the array of objects.

ex.

```
void main()
{
    Student s[3];
    for (int i=0 ; i<3 ; i++)
    {
```

Scanned by CamScanner

Object passed as an argument (object as f^n argument)

We can passed object as an argument. This may done in two ways

i) copy entire object into the calling f^n.

ii) passing address/creating reference into the calling f^n.

T1 → 5 Hours 15 min

T2 → 7 Hours 45 min

```
Struct time {
    int Hour;
    int min;
};
```

```
struct date
{ int dy;
  int mon;
  int Year; };
```

```
Struct rational {
    int nm;
    int dm;
};
```

eg. T1 : 5 Hour 35 minute

T2 : 7 Hour 45 minute

T3 : (SUM.)

T1		T2		T3	
Hour	min	Hour	min	Hour	min
5	35	7	45	13	20

```
class Time
{
    int Hour;
    int min;
    Public : void getdata (int h, int m)
        {
            hour = h ;   min = m ;
        }
    Void SUM ( Time &t1 , Time &t2 )   // call by Reference
        {
            min = t1.min + t2.min ;
            hour = min/60 ;
```

Scanned by CamScanner

```
min = min % 60;
hour = hour + t1.hour + t2.hour;
}
void display ()
{
    cout << hour << " Hour " << min << "Minute" << endl
} };
void main ()
{
    Time T1, T2, T3;
    clrscr ();
    T1. getdata (5,35);
    T2. getdata (7,45);
    T3. sum (T1, T2);
    cout << " T1: " ;
    T1. display ();
    cout << " T2: " ;
    T2. display ();
    cout << " T3: " ;
    T3. display ();
}
```

⇒ **Returning Object :-**

```
class Time
{
    int Hour;
    int min;
    Public : void getdata ( int H, int m )
    {
```

Scanned by CamScanner

```
    hour = h;    min = m;
}

Time sum (time &t)
{
    Time temp;
    temp.min = min + t.min;
    temp.hour = temp.min/60;
    temp.min = temp.min %60;
    temp.hour = temp.hour + hour + t.hour;
}
void display ()
{ cout << hour << " Hour " << min << " minute " << endl;
}};
void main ()
{
    Time T1, T2, T3;
    clrscr ();
    T1.getdata (5,35);
    T2.getdata (7,45);
    T3 = T1.sum (T2);
    cout << " T1: " ;
     T1.display ();
     cout << " T2 : " ;
    T2.display ();
    cout << " T3: " ;
    T3.display ();
}.
```

Scanned by CamScanner

Q WAP to perform arithmetic operator (+, -, x, /) of two complex number

Sol^n class complex

```
{
    float real ;
    float img ;
Public : void getdata ( float r, float i)
        { real = r ;   img = i ; }
void ~~display()~~ add ( complex &c1 , Complex &c2 )
 {
    real = c1. real + c2. real ;
    img = c1. img + c2. img ;
 }
void sub ( Complex &c1 , complex &c2 )
{
    real = c1. real - c2. real ;
    img = c1. img - c2. img ;
}
void mul ( Complex &c1 , Complex &c2 )
{
  real = c1. real * c2. real - c1. img * c2. img ;
  img = c1. real * c2. img + c2. real * c1. img ;
}
void div ( complex & c1 , complex &c2 )
 {
    int temp ;
  temp = ( c2. real * c2. real ) + ( c2. img * c2. img ) ;
  real = ( c1. real * c2. real + c1. img * c2. img ) / temp ;
  img = ( c1. real * c2. img - c1. img * c2. real ) / temp ;
}
```

Scanned by CamScanner

```
void display()
{
  cout << " complex number is : " << endl;
  cout << " real << " + i "<<imag << endl;
}};
void main ()
{
  clrscr ();
  complex c1, c2, c3, c4, c5, c6;
  c1. getdata ( 3.5, 3.5);
  c2. getdata (4.0, 2.5);
  c3. add (c1, c2);
  c4. sub (c1, c2);
  c5. mul (c1, c2);
  c6. div (c1, c2);
  c1. display();
  c2. display();
  c3. display();
  c4. display();
  c5. display();
  c6. display();
  getch ();
}
```

Scanned by CamScanner

DATE / /
PAGE

Friendly function (friend function) :- The private member can be accessed only through the member of class. It means the outside f^n donot have right to access any private member of any class.

ii) There are some situations where the two classes would like to access the common particular f^n. and some situations where we need to permit accessing of private member of the class through the outsider f^n.

NOTE :- For this purpose C++ introduce a new feature to make (called) friendly function.

⇒ A friendly f^n (friend function) have fully rights to access the private member of class.

⇒ To make the friend f^n we use the keyword **friend** in front of f^n declaration.

Syntax :-

friend (keyword) return_type function name (argument list);

Features of friend function :-

⇒ It has the fully right to access the private member.

Scanned by CamScanner

⇒ Its declaration statement is preceded by friend keyword.

⇒ Its defination does not have the membership identity label (class name of scope resolution operator). Bcz it is not a member of class.

⇒ In the calling statement it is called w/o object name & dot operator.

⇒ The member of the class in the body of friend f^n are accessed must with object name (A.x)

⇒ Generally the object is a argument of friend function

⇒ The declaration of friend f^n can be declared either in Public or private section of class w/o affecting.

eg.
```
class Average
{ int a; int b;
  Public :
    void getdata ( int m, int n)
      { a=m;  b=n ; }
      friend void average ( Average s);
};
void average ( Average s2 )
  { int t ;
    t = (s.a + s.b) / 2 ;
    Cout << " average " = " << t ;
  }
void main ()
  {
```

Scanned by CamScanner

```
Average P ;
P.getdata (5, 3);
  average (P);
}
```

eg.

```
class complex
{ float = real ;  float = img ; }
 Public :
      void getdata ( float r, float i )
      { real = r ; img = i ; }
      void display()
  {
   cout << "(" << real <<") + i (" << img << ")" << endl ;
}
   friend complex sum ( complex & , complex & ) ;
};
complex sum ( complex & c1 , complex & c2 )
     { ~~temp = temp~~ ;  complex temp ;
     temp.real = c1.real + c2.real ;
     temp.img = c1.img + c2.img ;
      return (temp) ;
     }
 void main ()
 {   complex c1, c2, c3 ;
   c1.getdata ( 2.3 , 5.2 ) ;
   c2.getdata ( 4.5 , 4.0 ) ;
   c3 = sum ( c1, c2) ;
   cout << " o:"
    c1.display ();
    c2.display ();
    c3.display ();
}
```

Scanned by CamScanner

Important NOTE :-

1) $\boxed{C3.\text{sum}(C_1, C_2)}$ → void complex :: sum (complex &c1, complex &c2)
member f^n
(w/o returning obj)

```
{ real = c1.real + c2.real ;
  img = c1.img + c2.img ;
}
```

2) $\boxed{C3 = C1.\text{sum}(C2)}$ → Complex complex :: sum (complex &c2)
member f^n
(with returning obj)

```
{ complex temp ;
  temp.real = real + c.real ;
  temp.img = img + c.img ;
  return (temp);
}
```

3) $\boxed{C3 = \text{sum}(C_1, C_2)}$ → complex sum (complex &c, complex &cc
↑
friend f^n

```
{ complex temp ;
  temp.real = c.real + cc.real ;
  temp.img = c.img + cc.img ;
  return (temp);
}
```

Q. WAP to compare the member of two different classes. (find the greatest)

Sol^n

```
class POR ;
class XYZ
{
  int m ;
  Public : void getdata ( int x )
           { m = x ; }
  friend int greatest ( XYZ &, POR & );
};
```

Scanned by CamScanner

```
class PQR
{
    int n;
    Public :
        void getdata ( int y)
            { n=y;
            }
    friend int greatest ( XYZ t , PQR s ) ;
};

int greatest ( XYZ t , PQR s )
    {
        if ( T.m > S.n )
            return T.m ;
        return S.n ;
    }
void main ()
{
    XYZ T ;
    PQR S ;
    int g;
    T.getdata ( 15 );
    S.getdata ( 10 );
    g = greatest ( T , S ) ;
}
```

we can also make the member f^n of one class as a friend to other class :

Scanned by CamScanner

Syntax:

```
friend return_type class_name :: function_name (argument list);
```

```
class POR;
class XYZ;
{
   int m;
   Public : void getdata (int x)
            { m = y; }
   int greatest ( POR s )
   {
         if (m > s.n)
            return m;
         else
            return s.n;
   }

};
class POR
   {
      int n;
      Public :
         void getdata ( int y)
            { n = y;
            }
   friend int XYZ :: greatest ( POR s );
   }
void main ()
{
   XYZ T;
   POR S;
   int g;
   T. getdata (15);
   S. getdata (10);
   g = T. greatest ( S );
}
```

Scanned by CamScanner

→ We can also make entire class friend of another class.

Syntax :- friend class-name ;

Memory allocation for objects :-

i) Earlier, we have learn the memory space for the object is allocated when they are declared, not when the class is specified.

ii) The above statement is partly true, actually the member f^n are created & placed in the memory space only once when they are defined as a part of class specification.

iii) Since all the concept belonging to the same member f^n so their is no separate space. ~~Since all~~ is allocated for the member function when the obj. is created.

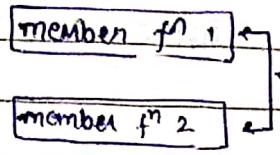
iv) The data member variables is allocated seperatly for each objects.

v) Seperate memory location for the object is essential bcz the separate data member will hold different data values for the different objects.

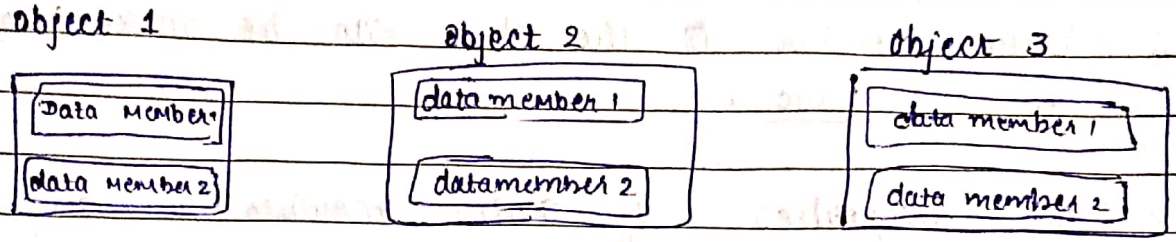
Scanned by CamScanner

DATE / /
PAGE

COMMON MEMORY Allocation



→ memory allocation is done at f^n defining in class.

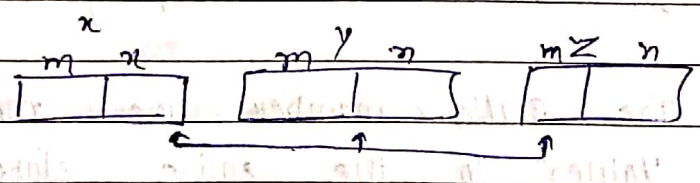


memory allocation for data members done at obj. creation.

eg.

```
class XYZ
{ int m;
  int n;
  Public : void getdata()
           { m=10 ; n=15 ;
           }
           void printdata()
           {
             cout<<"m="<<m<<" n="<<n<<endl;
           }
};
void main ()
{
  XYZ P,Q,R;
}
```

memory allocation for getdata() & printdata() is done at the time of class once specification.



Memory allocation for data member of class seperately for each obj is ~~at~~ done at obj. creation

Static data member & Static member f^n :-

Static data member :-

i) Data member of the class can be make or qualified as a Static.

ii) The properties of static member is similar to 'C' static variable.

iii) They are initialized to zero when the first object of its class is created.

iv) only one copy of static member is created for the entire class. & it is shared by all the objects of that class.

v) It is visible only within the class but its lifetime is entire program

vi) The static member are normally used to maintain values to the entire class.

eg.
```
class Test
{
    int number;
    static int count;
    public : void getnumber (int m)
        {   number = m;
            count ++;
        }
```

Scanned by CamScanner

```
void printdata()
{
    cout << " object value : " << number << " Total object count "
         << count << endl;
};  int test :: count;
void main()
{
    Test t1, t2, t3;
    t1. getnumber(10);
    t2. ———— " ———— (20);
    t3. ———— " ———— (30);
    t1. printdata();
    t2. ———— " ———— ;
    t3. ———— " ———— ;
}
```

t1	t2	t3	count
number	number	number	
10	20	30	~~0~~ ← initially ~~1~~ ~~2~~ 3

object value : 10 Total obj. count : 3
———— " ———— : 20 ———— " ————
———— " ———— : 30 ———— " ————

⇒ The static variables (data members) must be defined at outside the class as :

<u>syntax</u> : data-type class name :: static-member-name ;

or

datatype class-name :: static-member name = Values.

eg. $\boxed{\text{int test :: count ;}}$

<u>Static member f^n :</u>

i) like the static data members we can also have static member f^n.

Scanned by CamScanner

ii) A static member f^n restrict to access only other static member. f^n (Data or function).

iii) A static member f^n can be called using the class name instead of object name.

Syntax:- class.name :: static_member_f^n ();

eg.
```
class Test
{
    int code;
    static int count;
    Public : void setcode()
        {
            code = ++count;
        }
        void showcode()
        {
            cout << "object number : " << code << endl;
        }
        Static void Show count ()
        {
            count << "count : " << count << endl;
        }
};
int test :: count ;
void main ()
{
    test t1, t2;
    t1. setcode();
    t2. setcode();
```

Scanned by CamScanner

```
test :: Showcount();
test t3;
t3. showcode();
test :: Showcount();
t1. showcode();
t2. showcode();
t3. showcode();
}
```

t1 [1] t2 [2] count [3] t3 [3]

~~object number : 3~~ count : 2
count = 3
object number : 1
object number : 2
object number : 3

Constructor AND Destructor

Constructor :- i) A constructor is a member function which is used to perform automatic initialization of object at the time of object creation.

ii) It is a special function bcz its name is same as the name of the class belonging to it.

iii) Syntax :-

```
no return type
class_name ( argument list )
{   // body
}
```

iv) The constructor does not have any return type even not 'void'

Why constructor

```
int a;
int a = 5;   Explicit Initialization
int a(5);    implicit "
```

int a = 5 ⇒ a [5] 100

```
class XYZ
{ int m;
  int n;
pubic: void getdata(int x, int y)
       { m = x; n = y;
       }
XYZ P;
```

P: m n

Scanned by CamScanner

v) The constructor is automatically invoke / call when the object is created or declared.

vi) The constructor is defined in the public section of class.

Types of constructor :-

i) default constructor :- The constructor with no argument is called default constructor it means for calling this constructor we do not pass any argument when object is created.

ex.
```
Class Test
{
    int m;
    int n;
    Public : Test ()          // default constructor
        { m=10 ; n=15 ;
        }
    void putdata ()
    {
        cout<<" m="<<m <<" n="<<n ;
    }
};
void main ()
{ Test T;
  T. putdata();
}
```

2) Parameterized constructor (argumental Constructor) :-

We can pass the argument with the object at the time of object creation it means we have a constructor with parameters or argument.

ex

```
class Test
{   int m;
    int n;
    Public :   Test (int x, int Y)
               {   m=x ;  n=Y ;
               }
               void putdata ()
               {
                 cout << " m=" << m << " & n=" << n ;
               }
};
void main ()
{
    Test T (5,10) ;
      T. putdata () ;
}
```

T
m | n
5 | 10

called automatically

→ We can pass argument of any type except the class belonging to that constructor.

If we define constructor → illegal.

```
Test (Test P)
{ //body
}
```

Then it is illegal,

but we can create this constructor as

```
Test (Test &p) ==
{
  // body
}
```

we can make with the reference argument for the class belonging to it.

3) ~~Copy~~ overloaded constructor (constructor overloading) :-

we can make more than one constructor f^n in a class it means we have more than one f^n with same name but diff. argument list. so this mechanism is called constructor overloading.

ex

```
class Test
{ int m;
  int n;
}
```

```
class complex
{ float real;
  float img;
Public: Complex () // default
        { real = img = 0.0; }
        Complex ( float x ) // parameterized
        { real = img = x ;
        }
        Complex ( float r, float i )
        { real = r ; img = i; }
void sum ( complex &c1, complex &c2 )
{ real = c1.real + c2.real ;
  img = c1.img + c2.img ;
}
void display ()
{
  cout << " real << " + i " << img << endl ;
}
};
```

Scanned by CamScanner

```
void main ()
{   complex  C1 (2.8, 3.4), C2 (1.7), C3 ;
    C3. sum (C1, C2) ;
    cout << " C1: "; << C1. display ();
    Cout << " C2: " ;  C2. display () ;
    cout << " C3: " ;  C3. display () ;
}
```

3) <u>Copy constructor</u> :- i) It is used to initialize an object with another existing object of same class.

ii) we pass object as an argument of the constructor. A copy constructor cannot have the class object as a value for an argument so we use reference for that purpose.

```
Integer   I1(5,2) ;   // parameterised constructor
Integer   I2(I1) ;    // copy constructor
```

ex

```
class Complex
{
    float real ;
    float img ;
    public :  complex () { real = img = 0.0 ; }
              complex (float r, float i)
              { rear = r ; img = i ; }
              complex (complex & c)
              { real = c. real ;
                img = c. img ; }
              void sum (complex & C1 , complex & C2 )
              { real = C1. real + C2. real ;
                img = C1. img + C2. img ;
              }
```

Scanned by CamScanner

```
void display()
{ cout << real << " + i " << img << endl; }
};
void main ()
{ complex c1 (6.2, 6.3);
  complex c2(c1), c3;
  c3. sum (c1,c2);
  cout << "c1:"; c1. display();
  cout << "c2:"; c2. display();
  cout << "c3(sum)"; c3. display();
}
```

Constructor with default parameter / Argument :-

→ We can also create the constructor with default arg. It means we can set default value for the arguments at the time of constructor declaration / defination.

ex
```
complex (float r, float i = 2.5)   default value.
  { real = r;   img = i; }
```

Dynamic initialization of constructor :- The value for the argument of constructor can be passed at the time of execution or run.

ex
```
class Test
  { int m; int n;
    Public : Test ();
              { m=n=0;
              }
```

Scanned by CamScanner

```
Test (int x , int y)
  { m=x;  n=y; }
void putdata ()
{
   cout <<" m= " <<m <<"\n n= " << n ;
}};
void main ()
{ int a, b ;
   cout <<" Enter the value of a & b : \n" ;
   cin >> a>>b ;
    Test T (a, b) ;
    T. putdata ();
}
```

```
class test
{ int m;  int n;
public :
   Test ()
  { cout <<" Enter the value" ;
    cin >> m >>n ;
  }
  void putdata ()
  {
    cout <<"m= "<<m <<"\n n= "
                <<n ;
  }
  Void main ()
  {
    Test T;
    T.putdata () ;
  }
```

Constructor calling :- we know the constructor is automatically invoke or call at the time of object creation. The ~~passing of~~ calling of constructor can be either explicitly or implicitly.

Implicit :- Test T(5,2) ;

Explicit :- Test T = Test (5,2) ;

⇓

or Test T ;

T = Test (5,2) ;

2) **Dynamic constructor** :-

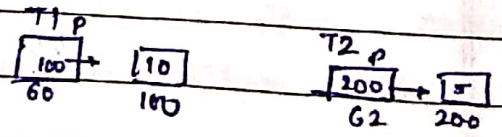
⇒ The constructor can also be used to allocate memory while creating objects it means the memory

allocation or the object at the run time. It called dynamic constructor of the object.

→ for the dynamic constructor we use <u>new</u> operator in the body of constructor

ex

```
class Test
{
    int *P ;
    Public :   Test ()
               {
                  P= new int ;
                  *P = 10 ;
               }
               Test (int x)
               {  P= new int ;
                  *P = x ;
               }
               void display ()
               {
                  cout << " value = " << *P ;
               }
};
void main ()
{
    Test T1 , T2(5) ;
    cout << " Object T1 " ;
      T1. display() ;
    cout << " object T2 ;
      T2. display () ;
}
```



object T1 :
Value = 10
Object T2 :
value = 5

Scanned by CamScanner

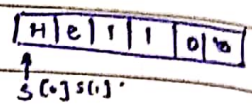
ex.

```
class stringtest
{
    char * name ;
    int length ;
    Public :  StringTest ()
              {
                  length = 0 ;
                  name = new char [length + 1] ;
              }
              StringTest (char *s)
              {
                  length = strlen (s) ;
                  name = new char [length + 1] ;
                  strcpy ( name , s ) ;
              }
              void display ()
              {
                  cout << name << endl ;
              }
              void join ( String Test &a , StringTest &b) ;
};
void stringTest :: Join ( String Test &a , string Test &b )
{
    length = a.length + b.length ;
    delete name ;
    name = new char [length + 1] ;
    strcpy ( name , a.name) ;
    strcat ( name , b.name) ;
}
```

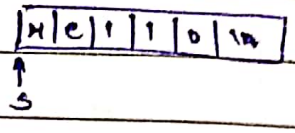
Scanned by CamScanner

```
int main()
{
    char *first = "Joseph";
    StringTest name1(first), name2("Lewis"), name3("legrange
                S1, S2;
    S1.Join (name 1, name 2);
    S2.Join ( S1, name 3);
    name1.display();
    name 2 . display();
    name 3. display();
    S1. display();
    S2. display();
    return 0;
}
```

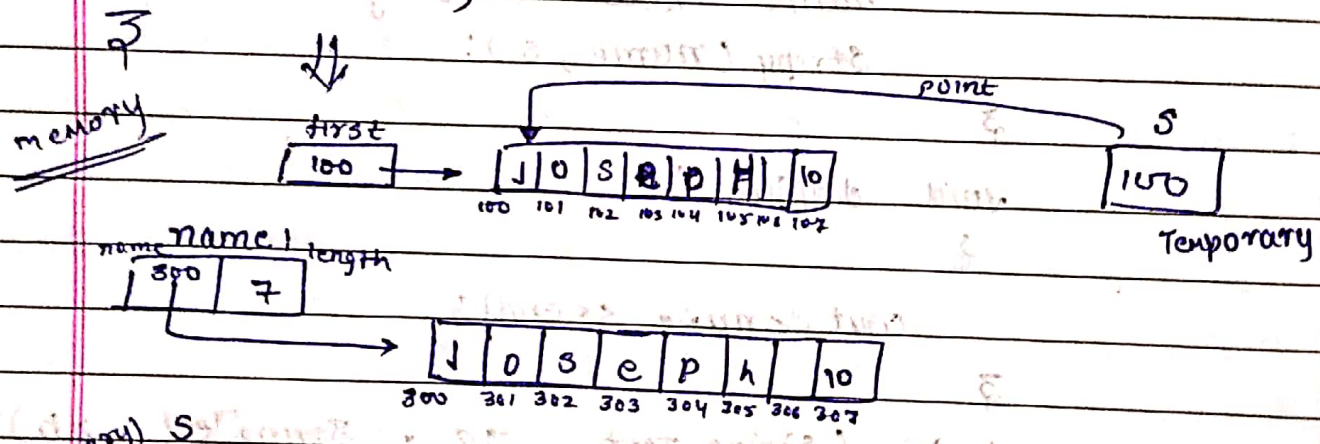
```
char s[6] = "Hello";
```



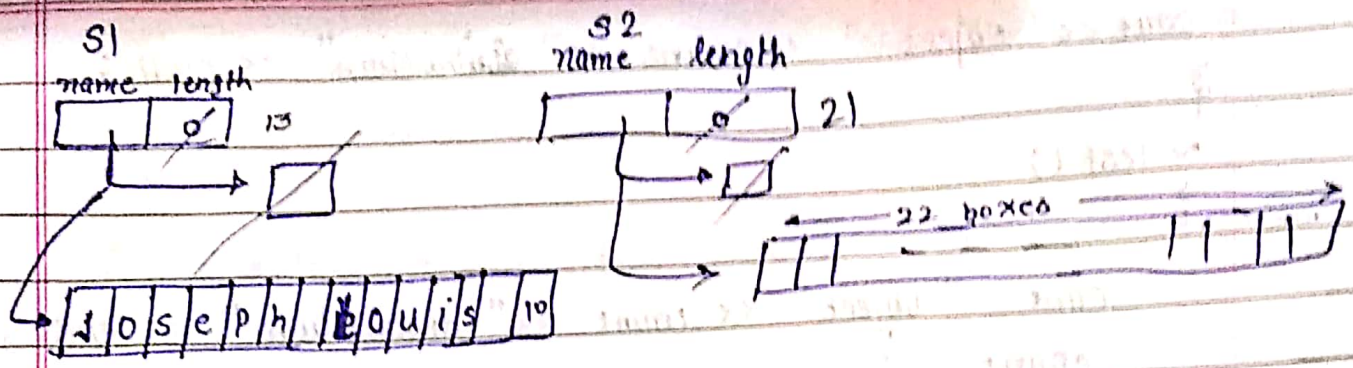

```
Char *s = "Hello";
```



memory



Scanned by CamScanner



Destructor :- It is used to destroy memory construct by the constructor.

It is a special member f^n used to destroy memory construct by the constructor.

→ The name of the destructor is same as the class name like constructor but it is preceded by (~) Tilde.

Syntax :-

```
~ class_name ()
{   // body
}
```

no argument

No Return

→ It is automatically Invoked when the Object goes out of scope.

→ It has no arguments and nor return type. (even not void).

eg.
```
class Test
{ static int count ;
  Public :  Test ()
            { Count ++ ;
```

```
cout << " object " << count << " construction " << endl ;
}
~ Test ()
{
    cout << " object " << count << " destruction " << endl ;
    count -- ;
}
};
int Test :: count ;
void main ()
{   Test T1 ;   constructor call for T1
    Test T2 ;   constructor call for T2
    {
        Test T3 ;   // constructor call for T3
    } // destructor call for T3
} //destructor call for T2 & T1
```

count

| 10 |

object 1 constructor
— 2 —
— 3 —
object 3 destructor
— 2 —
— 1 —

→ In the body of class we can define more than 1 constructor with diff. arg. list (overloaded constructor) But the destructor is used only one. Bcz it has no argument.

→ Destructor is define in the public section of class.

→ If the constructor is define as a dynamic constructor with the new operator

Scanned by CamScanner

Dummy constructor :- (do-nothing constructor)

```
XYZ()
{ }

XYZ P1(5), P2(5,7), P3;
```

then we must use delete operator in the body of destructor.

→ The destructor can be virtual but constructor not.

Operator Overloading (operator polymorphism)

1) The operator overloading is an imp. concept in C++. It is the type of polymorphism in which an operator is overloaded with special meaning for user defined datatype.
2) It permits us to perform operators on user defined datatype with same syntax i.e apply to inbuilt datatype.
3) for Ex- we can add two complex numbers (objects) in the syntax like $\boxed{C3 = C1 + C2}$ same as inbuilt type variables.
4) we can overload all the C++ operators except the following
   i) class member access operator (., .*)
   ii) scope resolution operator (::)
   iii) size operator (sizeof())
   iv) conditional operator (?:)

Defining operator overloading :- (Define overloaded operator f^n)

& To overload an operator, we just define a f^n

named operator with the operator symbol which being overloaded.

Syntax :-

```
return_type  operator  OP ( argument_list )
                 ↑          ↑
              keyword   operator symbol which being overloaded (+, -, *, < etc)
{
    // body
}
```

ii) Here the operator to collectively used as a f^n name.

iii) The operator f^n must be either member f^n or friend f^n. The basic difference between them is that friend f^n will have only one argument for unary operator and two for the binary operator, while a member f^n has no argument for the unary operator and only one for the binary operator.

	unary	Binary
member f^n	zero	one
friend f^n	one	two

iv) The operator f^n can be invoke like

a) unary operator :-

$\boxed{X \quad OP}$ or $\boxed{OP \quad X}$

X → object (operand), OP → operator

OP → operator, X → object (operand)

it interpreted as $\boxed{X.\text{operator } OP()}$ in case of member f^n and $\boxed{\text{operator } OP(x)}$ in case of friend f^n.

Scanned by CamScanner

b) Binary operator :- $\boxed{X \quad OP \quad Y}$

object 1 (operand 1) — operator — object 2 (operand 2)

⇒ It interpreted as $\boxed{X.\text{Operator } OP(Y)}$ (one argument) in case of member f^n and $\boxed{\text{operator } OP(X, Y)}$ (Two argument) in case of friend f^n.

⇒ $\boxed{X \quad OP \quad Y}$ ⇒ Here the first operand is always use a invoking object.

Ex. Complex operator -(Complex) ; // arithmatic subtraction (Binary minus)

friend Complex operator -(complex, complex) // Binary minus (subtraction)

Complex operator -() ; // unary
friend complex operator - (complex) ; // unary

Binary operator (+) overloading :-

```
Class complex
{   int real;
    int img;
Public:  complex () {}
         complex (int x) { real = x ; img = x ; }
         Complex (int r, int i)
         { real = r ;  img = i }
         complex operator + ( complex s ) ;
         void display ();
};
```

Scanned by CamScanner

```
complex complex :: operator + (complex &c)
  {  complex temp;
     temp.real = real + c.real ;
     temp.img = img + c.img;
     return (temp);
  }

complex complex :: display()
  {  cout << real << " + i" << img << endl;
  }
void main()
  {  complex c1(5,3); c2(8), c3;
     c3 = c1 + c2;
     cout << "c1" ;  c1.display();
     cout << "c2;" ;  c2.display();
     cout << "c3;" ;  c3.display();
  }
```

```
class complex
{  int real;
   int img;
   Public :  complex () {}
             complex (int x) { real = x ; img = x; }
             complex (int r, int i)
               { real = r; img = i; }
             friend complex operator + (complex &, complex &);
             void display();
};  complex operator + (complex &c1 , complex &c2)
      {
             complex temp;
             temp.real = c1.real + c2.real;
             temp.img = c1.img + c2.img;
             return temp;
      }
```

Scanned by CamScanner

```
complex complex :: display ()
{
    cout << real << " +i" << img << endl;
}
```

if the above operator (+) f^n used as a friend then we define as :-

```
⇒ complex operator + (complex & c1, complex & c2)
{ complex temp;
  temp.real = c1.real + c2.real;
  temp.img = c1.img + c2.img;
  return (temp);
}
```

calling	declaration
① c3.sum (c1, c2) →	void sum (complex &, complex &); member f^n
② c3 = c1.sum(c2) →	complex sum (complex &); // member f^n
③ c3 = sum (c1, c2) →	friend complex sum (complex &, complex &);
④ c3 = c1 + c2 →	complex operator + (complex &); // member f^n
	friend complex operator + (complex &, complex &)

Unary operator overloading :- (− sign)

```
class Test
{ int x;
  int y;
  Public:   Test () {}
            Test (int m, int n)
              { x = m; y = n; }
```

Scanned by CamScanner

```
void display()
{ cout << " x= " << x << " and " << " y: " << y << endl;
}

void operator - ()
{ x = -x ; y = -y ;
}
};

void main ()
{ Test T (5, -10) ;
  cout << " Before sign change " ;
  T.display () ;
  -T ;        //  T.operator -()
  cout << " After sign change T: " << endl ;
  T.display () ;
}
```

Ex.

```
class Test
{ int x;
  int y;
  Public: Test () { }
          Test (int m, int n)
          { x = m ;  y = n ;
          }
          void display ()
          {  cout << " x: " << x << " and y = " << y << endl
          }
          Test operator - ()
          { Test temp ;
            Temp.x = -x ;
            Temp.y = -y ;
            return (Temp) ;
          }
```

Ex

Scanned by CamScanner

```
{ Test T1(5,-10), T2;
    T2 = -T1;
    cout << " T1: " ; T1.display();
    cout << " T2: " ; T2.display();
}
```

Ex. (++ operator)

```
class Test
{ int x; int y;
  public: Test () {}
          Test (int m, int n)
          { x=m; y=n; }
          void display ()
          { cout << "x=" << x << "any=" << y << endl;
          }
          void operator ++ ()
          { ++x; ++y;
          }
void main ()
{ Test T(5,-10);
  ++T;
  T.display();
}
```

or

```
Test operator ++ ()
{ Test Temp;
  Temp.x = ++x;
  Temp.y = ++y;
}

Test T1(5,-10), T2;
for T2 = ++T1;
```

Ex. Overloading of Pre ++ & Post ++

```
class Test
{ int x;
  Public:
```

Scanned by CamScanner

```
Test () {}
Test (int m)
{ x = m; }
void display (char s){ cout << s << "x=" << x << endl; }
Test operator ++ ()                    // pre-increment
{ Test temp;
  temp.x = ++x;
  return (temp);
}
Test operator ++ (int)                 // post increment
{ Test Temp;
  temp.x = x++;
  return (temp);
}
};
void main
{
  Test T1(5), T2(7), T3, T4;
  T3 = ++T1;
  T4 = T2++;
  T1.display ("T1:");
  T2.display ("T2:");
  T3.display ("T3:");
  T4.display ("T4:");
}
```

T1: x = 6
T2: x = 8
T3: x = 6
T4: x = 7

In above program we use 'int' as a dummy argument in case of post-increment to create the difference between pre & post increment overloading f^n. It is already explained in the compiler

Binary operator overloading using friend function :-

i) We studied earlier the overloaded f^n may be either member f^n or friend f^n. The difference b/w them is only one argument is passed in case of member f^n while two argument must be passed explicitly in case of friend f^n.

ii) In most cases the result of member f^n or friend f^n is same but in some situations we would like to use friend f^n rather than member f^n.

iii) for Ex - $\boxed{B = A * 2}$

↳ In this situation we can use operator member f^n either member f^n or friend f^n w/o any problem

member f^n ⇒ A.operator * (2)
friend f^n ⇒ Operator * (A, 2)

But the statement

$\boxed{B = 2 * A}$
object int object

not work with member f^n so we must use as a friend f^n.

member f^n ⇒ 2.operator * (A) ✗
friend f^n ⇒ Operator * (2, A) ✓

Ex.
```
void main()
{   complex c1(2,5), c3, c4
    c3 = c1 * 2;
    c4 = 3 * c1;
    c1.display("c1:");
    c2.display("c2:");
```

Scanned by CamScanner

```
c3.display ("c3:");
c4.display ("c4:");
}
class complex
{ int real;
  int img;
  public : complex () { real = img = 0 }
  complex (int r, int i)
    { real = r; img = i; }
  void display( char *s)
    { cout << s << real << " +i " << img << endl;
    }
  complex operator * (int);
friend complex operator * (int , complex s);
};
complex complex :: operator * (int n)
  { complex Temp;
    Temp.real = real * n;
    Temp.img = img * n;
    return (Temp);
  }
complex operator * ( int n, complex c)
  { complex Temp;
    Temp.real = n * c.real;
    Temp.img = n * c.img;
    return (Temp);
  }
```

Scanned by CamScanner

Overloading:

```
class complex
{
    int real ; int img ;
public :    friend istream & operator >> (istream &, complex &);
            complex operator + (complex &) ;
            friend ostream & operator << (ostream &, complex &);
};
istream & operator >> (istream &din , complex &c)
  {     din >> c.real >> c.img ;
        return (din) ;
  }

ostream & operator << (ostream &dout , complex &c)
{
    dout << c.real << "+i" << c.img << endl ;
    return (dout);
}

complex complex operator + (complex &c)
  {
      complex temp ;
      temp. real = real + c.real ;
      temp. img = img + c.img ;
      return (temp) ;
  }

void main ()
{
    complex c1, c2, c3 ;
```

Scanned by CamScanner

```
cout << " enter the real & imy of C1 " << endl;
cin >> C1;
cout << " Enter the real & imy of C2 " << endl;
cin >> C2;
C3 = C1 + C2;
cout << " C1: " ; cout << C1;
cout << " C2: " ; cout << C2;
cout << " C3: " ; cout << C3; }
```

Eg.

```
class Test
{
    int m;
    int n;
    Public:  friend void operator >> (istream &, Test &);
             int operator == (Test &);
};
void operator >> (istream &din , Test &T)
    { cout << " Enter value of m & n " << endl;
        din >> T.m >> T.n ;
    }
int Test :: operator == (Test &T)
    { if (m == T.m && n == T.n)
            return (1);
      else
            return (0);
    }
void main ()
{
    Test T1, T2;
    cout << " Enter the content of T1: " << endl;
```

Scanned by CamScanner

```
cin >> T1;
cout << " Enter the content of T2 :" << endl;
cin >> T2 ;
if (T1 == T2)
{ cout << " Content of T1 & T2 are same" ;
}
else
    cout << " Content of T1 & T2 are different" ;
}
```

Type Conversion :- i) In the many programming situation we need to convert one datatype into another datatype. This process is called type conversion

ii) we already studied earlier the conversion b/w inbuilt datatype such as int to float, char to int etc

iii) Now we study the conversion in respect of user define datatype and this is divided into three types

a) Conversion from " Inbuilt data type into class type

b) conversion from class type to inbuilt data type

c) conversion from one class to another class type.

a) Conversion from inbuilt data type into class type :-

for conversion inbuilt type to class type we just define a constructor which taking an argument of inbuilt datatype. which want to convert into class type.

Scanned by CamScanner

for Ex- If we want to convert from int to class demo we define constructor like.

```
demo( int x)
{ // body
}
```

Ex

```
Class demo
{ int m;
  Public: demo (int x)
          { m=x ;
          }
          void display ()
          { cout << m ; }
};

void main ()
{ int t ;
  cout <<" Enter the value of t :" << endl ;
  cin >> t ;
  demo d=t ; // int → demo
}
```

demo d = t
demo int

Ex

```
class demo
{
  Char s[20];
  Public: demo (char b[])
          { strcpy (s, b) ;
          }
          int count ()
          { int c=0, i ;
            char ch;
```

Scanned by CamScanner

```
for (i=0 ; s[i] != '\0' ; i++)
{
    ch = toupper (s[i]) ;
    Switch (ch)
    {
        case 'A' :
        case 'E' :
        case 'I' :
        case 'O' :
        case 'U' :   c++ ;
    }
    return (c);
}
};

void main ()
{ demo d = " Engineering College ASMER " ;
    int count = d.count () ;
    cout << " Total variables = " << count ;
}
```

B) Conversion from class type to inbuilt type :-

i) To convert a class type into inbuilt type we first define operator typecasting f^n.

ii) This operator f^n have a name inbuilt datatype to which the class being converted.

for Ex — If we want to convert a class type demo into inbuilt type int, we define a Operator f^n like

```
                casting type
Operator int ()
    { // body
    }
```

→ This f^n is defined in source - class

Scanned by CamScanner

Class = Inbuilt ⇒ constructor

Inbuilt = Class ⇒ operator

DATE / /
PAGE

iii) There is ~~must~~ no return type & no argument.

iv) This f^n must be ~~with~~ return the integer type data as the f^n name.

Ex.
```
class demo
{
    int x;
    Public : demo (int m)
            { x = m ;
            }

            operator int ()
            {
                return (x*x);
            }
};
void main ()
{ demo d(10) ;      // Call constructor
  int s=d ;         // convert demo to int
  cout <<" sq no :" <<s ;
}
```

c) Conversion from one class to another class :-

For this purpose we can use one out of two approa
i) to define constructor in destination class.
ii) To define a operator f^n in source class.

for Ex - we have two class normal first and second.
and we want to convert first class type int second class type.

Scanned by CamScanner

```
first f ;
second s ;
s= f
```

i) **define constructor** :- Define a constructor in the destination class **second** which take an argument of type **first**.

```
second (first st)
{   // body
}
```

→ define in class **second**.

ii) **define operator f^n** :- we define the operator f^n named second to the source class first

```
Operator second ()
{   // body
}
```

→ define in class **first**

Scanned by CamScanner

```
first f;
second s;
s= f
```

i) define constructor :- Define a constructor in the destination class second which take an argument of type first.

```
second (first st)
{   // body
}
```
→ define in class second.

ii) define operator f^n :- We define the operator f^n named second in the source class first

```
Operator second ()
{   // body
}
```
→ define in class first

Inheritance :-

→ The inheritance is the mechanism of deriving a new class from an already existing class.

→ The inheritance provide idea of reusability it means the code written ones can be used again and again in number of new classes.

→ The new class can use all or some of the features or properties of already existing class and it can also define own features or properties.

Scanned by CamScanner

→ Here the new class aquired the property of old cl
but not vice-versa.

→ The old class & new class are called by the ne
pair

old → new

parent	→	child
super	→	sub
base	→	derived

Syntax for Inheritance :-

```
class new_class_name : visibility-mode old_class nam
{
    // body
};
```

Here new_class_name is a name of derived class and old_class_name is a name of base class. Here, 3 type of visibility mode in inheritance. public, protected & private

→ If the visibility mode is absent it means defau mode is used which is private mode

eg.

```
class B : public A
{ // body of class B
};
```

class B is derive from class A in public mode

```
class B : A
{ //body;
};
```

By default private

Scanned by CamScanner

⇒ In Inheritance we can derive a new class from already existing user defined class or inbuilt class.

Types of Inheritance :-

There are five types of Inheritance :-

i) Single level inheritance
ii) Multi level inheritance
iii) Multiple Inheritance
iv) Hierachical inheritance
v) Hybrid Inheritance

These five types of Inheritance are further categorised in three types of inheritance depending on visibility mode

i) public inheritance ii) protected inheritance
iii) private inheritance

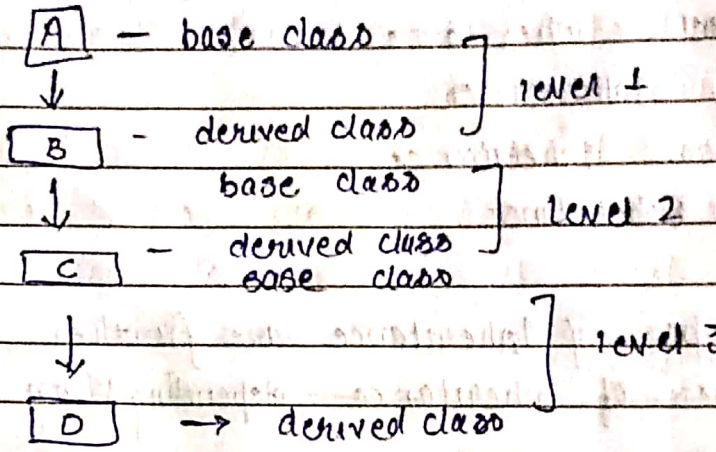
1) Single level :- In this type of inheritance one base class and one derived class is used.

[A] base / parent / super
↓
[B] derived / child / sub

```
class A
{ // body of A
};
class B: Public A
{ // body of B
}
```

Scanned by CamScanner

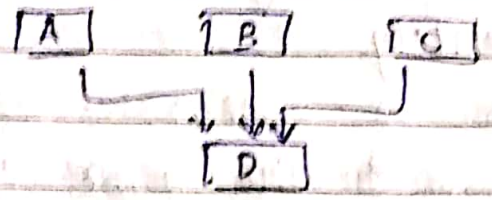
② **Multi-level inheritance:** In this type of inheritance More than 1 level is used. After level 1st one base class and one derived class are used. At the next level the derived class of 1st level become a base class and a new derived class.

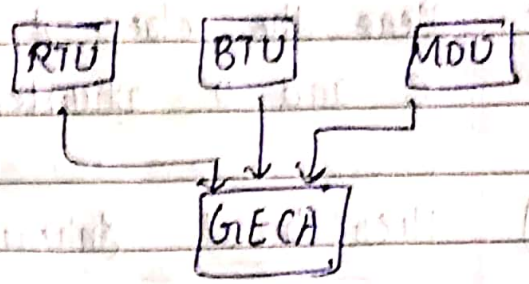



```
Class A
{ //body of A
};
class B : Public A
{ //body of B
};

Class C : Public B
{ //body of C
};

class D : Public C
{
    //Body of D
};
```

Scanned by CamScanner

3) **Multiple Inheritance :** This type of inheritance the class is derived from more than one class it means a derived class has more than one parent.






```
Class A
{ // Body of A
};

Class B
{ // Body of B
};
class C
{ // Body of C
};
Class D: public A, public B, public C
  { // Body of D
  };
```

Here the mode of every base class write seperately the mode of all base classes need not to be same.

```
class D: public A, protected B, private C
{ // Body
};
```

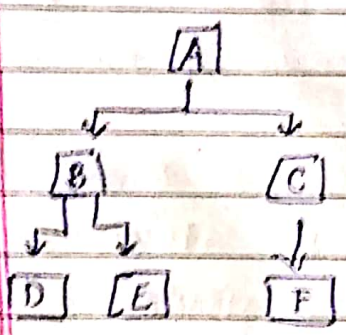
If the mode is absent we take default private mode.

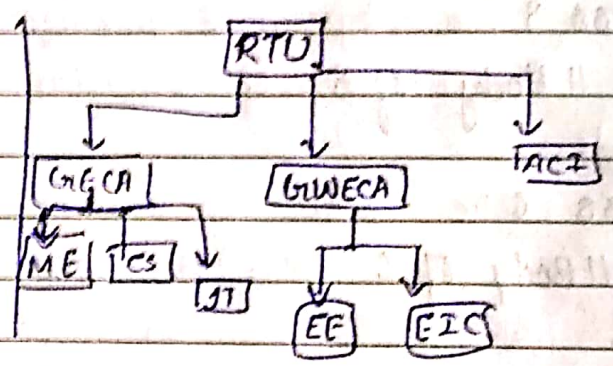
Scanned by CamScanner

```
class D : public A, B, C
{ //Body
};
```

→ Here the class A inherited publically but the class C and D inherited privately.

4) <u>Hierachical Inheritance</u>:- In this type of Inheritance the base class can inherited in more than 1 derived class at level one and the derived class are further derived class inherited in new derived by at next level and so on.





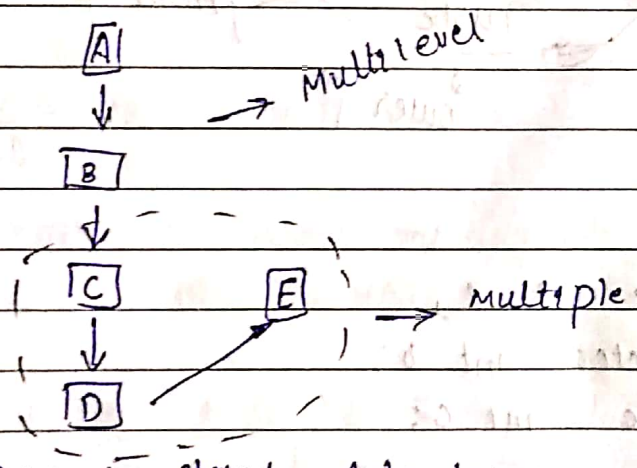

```
class A
{ //Body of A
};
class B : public A
{ //Body of B
};
class C : public A
{ //Body of C
};
```

Scanned by CamScanner

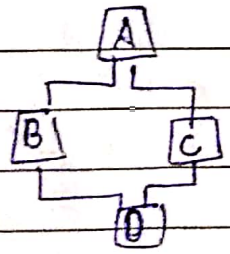
```
class D : public B
{   // Body of D
};

class E : public B
{   // Body of E
};

class F : class C
{   // Body of F
};
```

(5) **Hybrid Inheritance :-**

Combination :-



Diamond shaped Inheritance



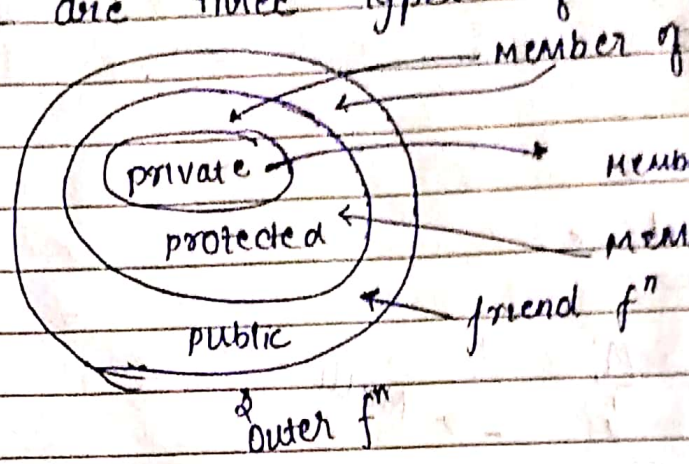

```
class A {
    // Body of A
};
class B : public A
{   // Body of B
};
class C : public A
{   // Body of C
};
```

Scanned by CamScanner

```
class D: public B, public C
{ // body of D:
};
```

Access Specifier of Member of class:

There are three types of Access Specifier




```
class A
{ private int a;
  protected int b;
  public int c;
};
class B: public A
{ member function ()
};
void main ()
{ outer function ();
}

friend function of class B;
```

→ The private members of class can be accessed only member of own class, friend function, member of friend class.

→ The protected class/member of the class can accessed only through the member of f^n of own class, friend f^n, member of friend class, and member of sub class.

→ The public member of class can be accessed through the all type of function even outer function.

Visibility Mode of Inheritance :-

→ We can inherit a class into new class through three types of visibility mode public, private & protected.

Base class A

Private
protected
public

Scanned by CamScanner

DATE / /
PAGE

Single level :-

eg.

```
class base
{ protected : void show()
              {
                cout << "Hello" ;
              }
};

class derived : public base
{ };

void main ()
{ derived d;
  d.show ();
}
```

it becomes protected

o/p error

```
class A
{ public : void show ()
          { cout << " i from A";
          }
};

class B : public A
{ public : void showB()
           { cout << " i from B";
           }
};

void main ()
{
  B b;
  b.showB();
  b.showb();
}
```

if protected / private

if protected & private → error

o/p
i from A
i from B

o/p error

eg.

```
class A
{ public : void show_A ()
           { cout << " i from A" ;
           }
};

class B : protected A
{ public : void show_B()
           {
             show_A ();
             cout << " i from B" ;
           }
};

void main ()
{ B b;
  b.show_B();
}
```

i from A
i from B

Scanned by CamScanner

eg. Class A
{ protected : void show_A ()
{ cout << " i from A \n" ;
}
};

Class B : public A
{ public : void show_B ()
{ cout << " i from B" }
}
};
void main ()
{ B b ;
b . show_A () ; // error
b . show_B () ;
}

eg. Class A
{ int a ; → if protected
public : void input_A ()
{ a = 10 ;
}
void print_A ()
{ cout << " a = " << a << endl ;
}
};

~~Class~~ B : public A
{ int b ;
public : void input_B ()
{ b = 15 ; → if protected input_A () ;
}
void print_B ()
{ cout << " b = " << b ; → if protected input_B () ;
}

Scanned by CamScanner

```
};
void main ()
{   B t;
    t.input_B();  ---> If protected error
    t.input_A();
    t.print_A();  ---> If protected error
    t.print_B();
}
```

a = 10
b = 15

eg.

```
class A
{ int a;
  protected : void input_A(int x)
              { a=x;
              }
              void print_A()
              { cout<<"a=" << a << endl;
              }
};

class B : Public A
{ int b;
  Public : void input_B(int m, int n)
           {  input_A(m);
              b=n;
           }
           void print_B()
              { print_A();
                cout<<" b=" <<b<<endl;
              }
};
void main()
{   B t;
    t.input_B(10,15);
    t.Print_B();
}
```

Scanned by CamScanner

Multilevel Inheritance :-

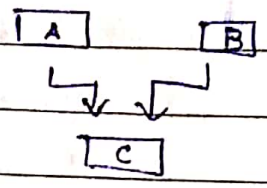
A
↓
B
↓
C

eg.

```
class A
{
    public : void show_A ()
            {cout << " Hello from A\n" ;
            }
};

class B : Public A        ---> protected
{ Public : void show_B()
            { cout << " Hello from B" ;
            }
};

class C : public B        ---> protected
{ Public : void show_C()
            { cout << " Hello from C" ;
            }
};

void main ()
{ C c ;
    c.show_A () ;
    c.show_B () ;
    c.show_C () ;
}
```

show_A() ; show_B() ; → " error if protected

Multiple Inheritence :-



eg.

```
class A
{   Public : void show-A ()
             {cout <<" Hello from A \n" ;
             }
};
class B
{ public :   void show-B ()
             {cout <<" Hello from B \n" ;
             }
};
class C : public A , public B
{ public : void show-C ()
           { cout << " Hello from C \n" ;
           }
};
void main ()
{   C t ;
    t. show-A ();
    t. show-B ();
    t. show-C ();
}
```

eg.

```
class Base
{ protected : int b ;
              void input-base ()
              { cout <<" Enter base \n" ;
                cin >> b ;
              }
```

Scanned by CamScanner

```
void show_base ()
{
    cout << "b= " << b << endl;
}

};

class exponent
{

  protected : int e;

        void input_exp ()
        { cout << " Enter exponent \n" ;
          cin >> e;
        }

        void show_exp ()
        { cout << " exponent is " << e << endl;
        }
};

class power : public Base, public exponent
{ int po;

    public :    void input ()
                {

                    input_base ();
                    input_exp ();
                }

        void show_po()
        { int i;
          po = 1
          for (i=1; i<= e; i++)
                po = po * e;
          cout << " power po = " << po;
        }
};
```

Scanned by CamScanner

```
void main ()
{
power p;
p.input();
p.show();
}
```

Ambiguity in multiple Inheritance :-

In the multiple Inheritance when two base classes having same name for the f^n & data ambiguity arises when they are used in the derived class.

ex- we have two classes A and B having same f^n signature for the f^n like show (); Both the classes are inherited in class C now accessing show () f^n in the class C creates confusion as to which so we want to refer

Resolution of ambiguity :-

To resolve the ambiguity we use scope resolution (::) with class name to f^n for accessing a f^n of individual class. It means we access the show() of class A as A :: show () and show() of class B as B :: show ()

eg.

```
Class A
{
    public : void show ()
            {
                cout << " Hello from A \n" ;
            }
};
class B
{
    public : void show ()
            {
                cout << " Hello from B \n" ;
            }
};

class C : public A, public B
{
    public : void show ()
            {
                cout << " Hello from C \n" ;
            }
};
void main ()
{
    C t ;
    t. show ();
    t.A :: show ();
    t.B :: show ();
}
```

function overriding :- In a single inheritance when a base class derived class having a same signature for the member f^n it is called f^n overriding. It means that member f^n has the two version base class version & derived class version. Simply it means the member f^n of same class re-defined (override) in derived class

when we call that f^n through the derived object it always goes to access derived class version But their is no ambiguity.

eg.
```
class A
{ public :  void show ()
            { cout << " Hello from A" ;
            }
};

class B : class A
{ public : void show ()
           { cout << " Hello from B" ;
           }
};
void main ()
{ B b;
  b.show ();
  b.A :: show ();
}
```

redefined

Hello from B
Hello from A

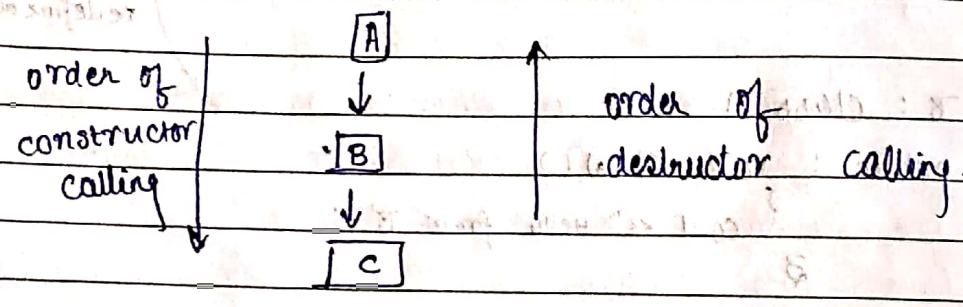
If we want to access the version of base class we use scope resolution operator with the class name

Inheritance :-

Constructor & destructor in Inheritance :- If Constructor & destructor are present in both base & derived class the invoking of constructor is in the sequence of classes implement in inheritance.
It means the base class constructor called first then derived class constructor.

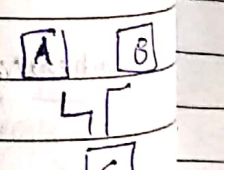
Scanned by CamScanner

Calling a constructor of base class first allows base class to properly setup its data member so that they can be used in derived class.
The destructor is called in the reverse sequence of the constructor invoking. It means destructor of derived class called first then destructor of base class.



⇒ In the multiple inheritance, constructor of all base classes invoked first then derived class constructor called.
Order of constructor calling depends upon order of How the base classes inherited.

```
class C : public A , public B
{
};
```



In above example constructor of class A called then constructor of class first then finally constructor of class C is called.

→ The destructor of derived class called first & destructor of base classes called from last towards first in sequence.

ex.
```
class first
{ Public : first ()
           {
           cout <<" Hello from constructor first \n";
           }

           ~first ()
           {
           cout <<" Bye from destructor first \n"}
};
class second : public first
{ public : second ()
           {
           cout <<" Hello from constructor second \n";
           }

           ~second ()
           {
           cout <<" Bye from destructor second \n";
           }
};
void main ()
{
    second s;
}
```

O/P

Hello	from	constructor	first
Hello	"	"	second
Bye	"	destructor	"
Bye	"	"	first

ex
```
class A
{ public : A ()
           {
           cout <<" Hello from A \n";
           }

           ~A ()
           {
           cout <<" Bye from A \n";
           }
};
```

Scanned by CamScanner

```
class B : ~~public A~~
{ public : B()
           { cout << " Hello from B \n" ;
           }
           ~B()
           { cout << " Bye from B \n" ;
           }
};

class C : public A , public B
{ public : C()
           { cout << " Hello from C \n" ;
           }
           ~C()
           { cout << " Bye from C \n" ;
           }
};

void main
{ C t ;
}
```

O/P

Hello	from	constructor	A
Hello	from	"	B
"	"	"	C
Bye	"	destructor	C
"	"	"	B
"	"	"	A

Scanned by CamScanner

eg. parameterised / argumental constructor in inheritance :

```
class first
{   int f ;
    public : first (int x)
             { f = x ;
               cout << " constructor called of first " ;
             }

             void f_show ()
             {  cout << " f = " << f << endl;
             }
};
class second : public first
{   int s ;
    public : second (int m, int n) : first (m) ,//s(n)
             {  s = n ;
                cout << " constructor called of second " ;
             }

             void show ()
             {  cout << " s = " << s << endl;
             }
};

void main ()
{   second t (10, 15) ;
    t.f_show ();
    t.show ();
}
```

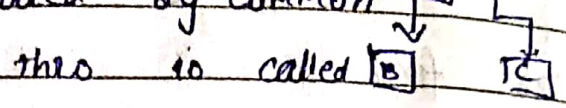
OR

t

f	s
10	15

Scanned by CamScanner

Hirearchical Inheritance :- when the no. of class are inherited by common base class is called, then this is called Hirearchal Inheritance.



→ When any change in the base class then this will rep. all derived class

```
class university
{ protected :    char uname [30] ;
    public :    university ()
                { strcpy( uname," Biraner technical University ";
3;              }

class college : public university
{   char    cname [30];
    Public :  college ()
              { strcpy (cname, " GNECA" );
              }

              void show-college ()
              { cout << " college Name :" << cname << endl;
                cout << " Affiliated university name :" << uname << endl
              }
3;

class college 2 : public university
{ ~~pub~~  char  cname [30];
    public :  college 2 ()
              {  strcpy ( cname, " GWECA" );
              }

              void show college ()
              {
```

Scanned by CamScanner

```
cout<<" college name:" << cname << endl;
cout << " Affiliated university name:" << uname << endl;
};
void main ()
{
    college c1;
    college 2 c2;
    C1. show-college ();
    C2. show-college2 ();
}
```

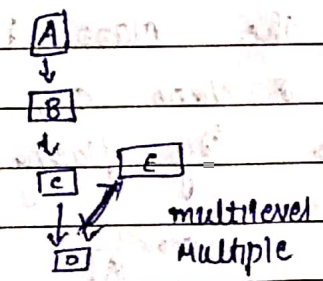
C1	
uname	cname
BTU	GECA

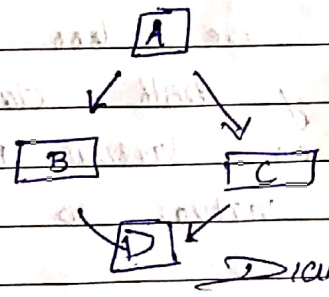
C2	
uname	cname
BTU	GWECA

O/P

```
college name : GECA
Aff. to university name : BTU
college name : GWECA
Aff
```

Hybrid Inheritance :- In some situation we need to implement the combination of diff type of inheritance this type is called Hybrid Inheritance



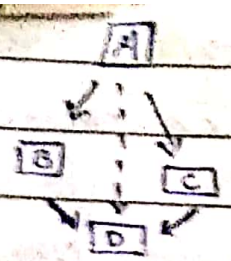


a) Hierachical
b) multiple

The ambiguity in Hybrid / Diamond Inheritance :-

Virtual Base class :- consider the situation where class A is inherited by two classes By classes B & C at first level (Hierachical Inheritance) & further the class B & class C both are inherited by class D (common) at the second level. (Multiple Inheritance)

Scanned by CamScanner

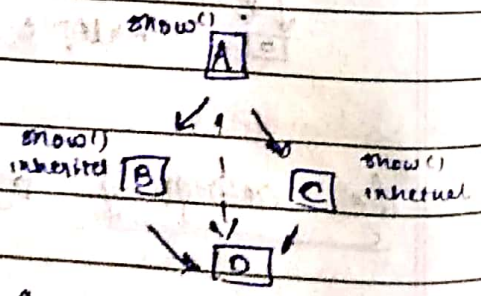


as seen in the diag that the data member & f^n of class A are inherited twice to the class D. one through the class B & second through the class C.

→ when any data member or f^n of class A accessed by an object of class D, an ambiguity arises as to which data or f^n would be called. one inherited through the B & another copy inherited through class C.

So the Compiler get confused & it display an error.

→ <u>Resolution of ambiguity</u> :- To resolve this ambiguity we use the concept of virtual Base class. when the class A is inherited by both class B & class C, It is declared as virtual Base class by placing the keyword virtual as




```
class B : virtual public A
{
};

class C : ~~virtual~~ public virtual A
{
};
```

→ virtual can be written before or after the mode

When the Base class is specified as a virtual base it can be treated as indirect base class it ensure that only single copy is created for the member of class A

→
```
class A
{ public : void show_A ()
          { cout <<" Hello from A \n" ;
          }
};

class B : public virtual A
{ public : void show_B()
          { cout <<" Hello from B \n" ;
          }
};

class C : public virtual A
{ public : void show_C ()
          { cout <<" Hello from C \n" ;
          }
};

class D : public B, public C
{ public : void show_D ()
          { cout <<" Hello from D \n" ;
          }
};
void main ()
{ d ;
  d. show_A ();
  d. show_B ();
  d. show_C ();
  d. show_D ();
}
```

Scanned by CamScanner