CO1: Differentiate Procedural Oriented programming and object oriented programming (K2)

Introduction to OOP

→ Programming Paradigams

→ Feature of OOP

→ datatypes

→ Variables, constants, Operators

→ decision statements and control structures

→ Arrays, name space, default arguments, constant arguments, Inputing default augments, reference arguments.

Compiler:- compilers are softwares, which are used to translate the source code to target code for.

Ex:- C compiler translates C code to binary code

ctt compiler translates c++ to C-code

Java compiler translates Java to byte codes.

→ compilers, Interpreters and Assemblers are Language translators.

ASCII :- (American standard code for Information Interchange)

a-97, b-98, c-99, d=100- - -

A-65, B-66, C-67, D=68 - - -

There are 255 ASCII values.

Datatypes in c++

TOKens:- Smallest Individual units in a program are called TOKens. c++ has the following tokens

1. Keywords
2. Identifiers

3. Conitants

4. strings

5. opuators

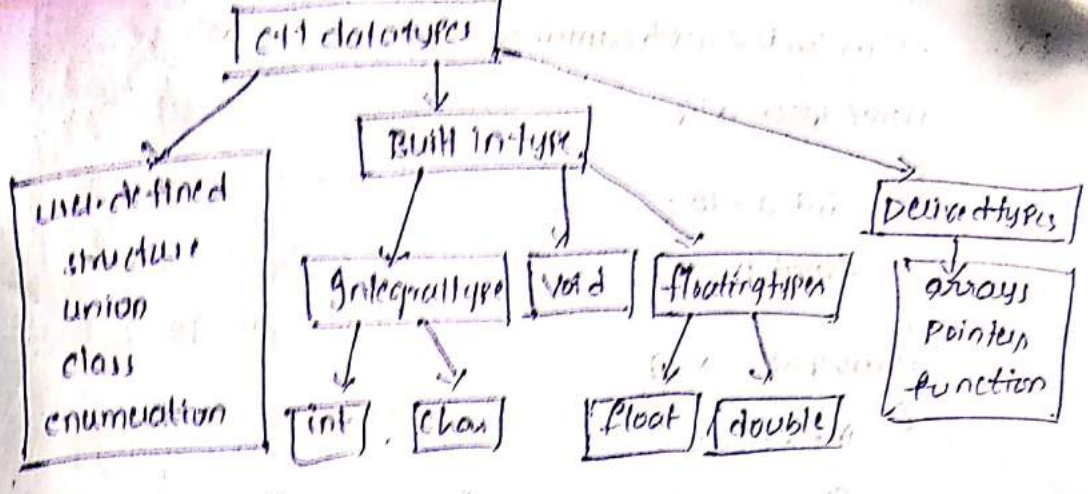
Keywords:- These are the reserved words; we can also use these reserved words in c and some reserved words are exclusively used with c++

Ex:- asm, auto, break, case, catch, char, class, const, continue, default, delete, do, double, else, enum, extern, float, for, friend, goto, if, inline, int, long, new, opuator, Private, Protected, Public etc

Identifiers:- It refers to the name of variables, functions, arrays, Pointers, Classes etc created by the programmer.

Identifiers are common to any programming Languages. The given rules are common to c and c++ for Identifiers for naming an Identifier

1. only alphabets, digits and underscores are permitted
2. The name cannot start with digit
3. like C, c++ is also case sensitive that means upper case alphabets are not equivalent to lower case alphabets
4. keywords cannot be used as a variable name.



Operators in c++

Both 'c' and 'c++' suppots the following opeators

1, Arithmetic opeators +, -, *, /, %.

2, Relational " >, <, >=, <=, ==, !=

3, logical " &&, ||, !

4, Inacement and decument ++, --

5, Assignment opeators =, +=, -=, /=, %=, *=

6, Bitwise opeators &, |, ^, >>, <<
(insertion, Extraction)

7, special opeators . (membership opeators)

8, Conditional opeators ?:

c++ Suppots all opeators in c and also Includes:

<< - overloaded as Insertion opeators used with cout

>> - overloaded as extraction opeats used with cin.

:: - Scope Resolution opeators

delete - memory release opeators

new - memory allocation opeators.

Namespace- It is one of the new features Introduced by the ANSI c++.

To avoid Name clashes namespaces are useful in c++

Syntax:-

namespace name

{

- - -

- - -

Ex:-

```
#include<iostream.h>
namespace abc
{
    int a=10;
    float b=3.2;
}
namespace xyz
{
    int a=20;
    float b=6.3;
}
int main( )
{
    cout<<"a+a "<<endl;
    cout<<abc::a+xyz::a<<endl;
    cout<<abc::b+xyz::b<<endl;
}
```

In c language we are using malloc( ), calloc( ), realloc( ) functions which are defined in stdlib.h (or) alloc.h

Memory management operator.

These functions are used for dynamic memory allocation that means runtime memory allocation.

For dynamic memory deallocation we are using free function in c language.

In c++ we consider new operator for creating a memory dynamically.

we can use delete operator for releasing the content of the memory dynamically.

Decision statements in c++

In c++ we can use decision statements same as in c-programming. They are

1) Simple if
2) If-else

3. Nested if else
4. else if laddu
5. Switch case
6. Break
7. default

108 header files in c++
27 header files in c

Biggest of three numbers using if else laddu.

```
#include <iostream.h>
#include <stdio.h>
#include <conio.h>
void main()
{
    int a,b,c;
    cout<<"enter a,b,c values";
    cin>>a>>b>>c;
    if(a>b && a>c)
     cout<<"a is big";
    else if(b>c)
     cout<<b<<"is big";
    else
     cout<<c<<"is big";
   getch();
}
```

① Write a program in c++ by using the following data.

An Insurance policy base premium is set at 75.32 Rupees. According to the age of a driver the premium will be decided. if driver age is less than 21 then 300 Rupees fine will be enforced on him. If age is in between 21 and 26 premium is 100 Rupees. If the age is in between 26 and 50 the Premium will be 50 Rupees.

The minimum premium is 75.32 according to the age given before the premiums are added to the given amount.

④ An employee is having basic salary which is Inputting from the Keyboard based on the basic salary the below calculations are to be made.

If, basic <= 5000
- Hra = 30% of basic
- Da = 40% of basic
- PF = 10% of basic
- Tax = 0% of basic

2, basic > 5000 and <= 10000
- hra = 28% of basic
- DA = 36% of "
- PF = 13% " "
- Tax = 0% " "

3, basic > 10000 and <= 20000
- hra = 26% of basic
- DA = 34% of "
- PF = 12% " "
- Tax = 3% " "

4, basic > 200,000 and <= 400000
- hra = 25% of basic
- Da = 32% of "
- PF = 10% " "
- Tax = 5% " "

5, basic above 40000
- hra = 22% of basic
- DA = 30% of "
- PF = 8% of "
- Tax = 10% " "

netsalary =

(Basic + hra + DA) - (PF + Tax)

print basicsalary
- hra
- DA
- PF
- Tax
- netsalary on the Screen.

```
#include <iostream.h>
int main()
{
    int age;
    float pre = 75.32;
    cin>>age;
    if (age <= 20)
    cout<<"the person is fined with 300 rupees";
    if (age > 20 && age <= 26)
    {
        pre = pre+100;
        cout<<"premium is "<<pre;
    }
```

```
if(age>26 && age<50)
{   pre=pre+50;
    cout<<"premium is "<<pre;
}
return 0;
}
```

(2)
```
#include<iostream.h>
int main()
{   int b;
    float h,d,P,t,n;
    cout<<"enter b value";
    cin>>b;
    if(b<=5000)
    {
        h=0.3*b;
        d=0.4*b;
        P=0.15*b;
        t=0*b;
    }
    else if(b>5000 && b<=10000)
    {   h=0.28*b;
        d=0.36*b;
        P=0.13*b;
        t=0*b;
    }
    else if(b>10000 && b<=20000)
    {   h=0.26*b;
        d=0.34*b;
        P=0.12*b;
        t=0.03*b;
    }
    else if(b>20000 && b<=40000)
    {   h=0.25*b;
        d=0.32*b;
        P=0.1*b;
        t=0.05*b;
    }
```

```
else
{   h= 0.22* b;
    d = 0.3 * b;
    P= 0.08* b;
    t= 0.1 * b;
}
n= (b+h+d) - (p+t);
cout<< "hra is "<<h;
cout << "\nda is "<< d;
cout << "\npf is "<< p;
cout << "\nter is"<< t;
cout << "\nnet salary is "<<n;
return 0;
}
```

③ Program. 4,5,6 – summer
7, 8, 9 – Rainy. using Switchcase.
10, 11, 12. winter
1, 2, 3 – Autumn.

```
#include<iostream.h>
void main
{
    int ch;
    ~~Print("enter char value");~~ cout<<"enter ch value";
    ~~scanf("%d",&ch);~~ cin >>ch;
    Switch (ch)
    {
     case 1 : ~~cout<<"Autumn"~~;
              ~~break~~;
     Case 2 : ~~cout<<"Autumn"~~;
     case 3 : ~~break~~;
     ~~case 3~~ cout << "Autumn";
              break;
     Case 4 : ~~cout<<"Summer"~~;
              ~~break~~;
```

```
case 5:
cout<<"summer";
break;
case 6: cout<<"summer";
        break;
case 7 : cout<<"Rainy";
        break;
case 8: case
case 9 : case
    cout<<"Rainy";
     break;
case 10:
case 11:
case 12:
    cout<<"winter";
    break;
}
getch();
}
```

1. counter controlled
2. sentinal control (or) conditional control

Counter Controlled :- In c++ the no. of Iterations of a loop are predefined in the program itself.

Default Arguments :-

c++ allows us to call a function without specifying all its arguments in such cases function assigns default value to the parameter which doesnot have a matching argument in the function call.. default values are specified when the function is declared. The compiler looks at the prototype to see how many arguments a function uses and alerts the program for possible default values

Ex:-

1. int fun(int a, int b, int c = 10); // legal
2. int fun(int a, int b = 5, int c = 10); // legal

3. int fun(int a=3, int b=5, int c=10); // legal

4. int fun(int a=3, int b=5, int c); //illegal

5. int fun(int a, int b=5, int c); //illegal

6. int fun(int a=5, int b, int c); //illegal

[default argument contain in prototype also definition]

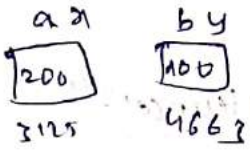
⇒
```
#include<iostream.h>
int sum(int a=10, int b=20, int c=30);
void main()
{
    int a,b,c,s;
    cout<<"enter a,b,c";
    cin>>a>>b>>c;
    s=sum(a,b,c);
    cout<<"sum without usage of default arg"<<s<<endl;
    s=sum(a,b);
    cout<<"sum using only one default arg"<<s<<endl;
    s=sum(a);
    cout<<"sum using 2 default arg"<<s<<endl;
    s=sum();
    cout<<"using default arg"<<s<<endl;
    getch();
}
int sum(int x, int y, int z)
{
    return(x+y+z);
}
```

C++ call by Reference:- This method of passing arguments to a function copies the reference of an argument into the formal parameter inside the function the reference is used access the actual argument used in the call. This means that changes made to the parameter effect the passed argument.

To pass the value by reference argument reference is passed to the function just like any other value. so, accordingly we need to declare the function parameters as reference types as in the following function which exchanges the values of the two integer variables pointed to by its arguments.

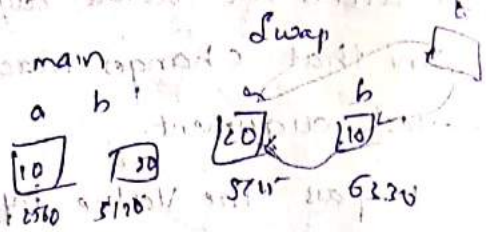
Program

```
void swap(int &x, int &y)
{
    int temp;
    temp= x ;
    x= y;
    y=temp;
    return ;
}
int main( )
{
    int a=100, b=200;
    cout<<"Before swap a="<<a<<" b="<<b<<endl;
    Swap (a,b);
    Cout<<"After swap a = "<<a<<" b = "<<b<<endl;
    return 0;
}
```



call by value

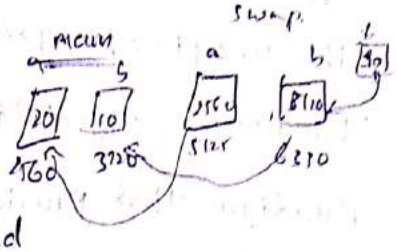
```
Void swap (int a, int b)
{
    int t;
    t = a
    a = b
    b = t;
}
void main()
{
    int a, b;
    cout << "enter a, b";
    cin >> a >> b;
    cout << "before swap \n";
    cout << "a = " << a << "b = " << b << endl;
    Swap(a, b);
    cout << "\n After swap \n";
    cout << "a = " << a << "b = " << b << endl;
}
```



Call by refernce:-

```
Void swap ( int *a , int *b)
{
    int t;
    t = *a;
    *a = *b;
    *b = t;
}
void main()
{
    int a, b;
    cout << "enter a, b",
    cin >> a >> b;
    cout << "before swap \n";
    cout << "a = " << a << "b = " << b << endl;
    Swap (&a, &b);
```

```
cout<<"In After swap\n";
cout<<"a="<<a<<"b="<<b<<endl;
}
```



Constant arguments :-

The constant variable can be declared using const this keyword makes variable value stable. The constant variables should be initialized while declaring

Syntax:-
```
int strlen(const char *p);
int length(const string &s);
const int x; //illegal
const int x=10; // legal
```

The qualifier const tells the compiler that function should not modify the argument, The compiler will generate error when this condition is violated. This type of declaration is significant only when we pass arguments by reference (or) pointer

```
#include<iostream.h>
int main()
{
    int min(const int a=8, int b=10);
    int a=12, b=45;
    b=min(a);
    cout<<"In a="<<a<<"b="<<b;
    return 0;
}
int min(const int j, int k)
{
    //j++;
    k++;
    cout<<"n j="<<j<<"k="<<k<<endl;
    if(k<j)
        return k;
    else return j;
}
```

Introduction to oop in c++:-

The major purpose of C++ programming is to introduce the concept of object orientation to the C-programming Language. object Orientation programming is a Paradigm that provides many concepts such as datamining, class, objects

Object:- An entity that has state and behaviour is known as Object.

Ex:- chair, open, book, table etc.

It can be physical & logical

Class:- Collection of objects is called class. It is logical Quantity.

Inheritance:- When one object acquires all the properties and behaviours of parent object it also provides code reusability. It is also used for run time polymorphism

Polymorphism:- When one task is performed by different ways which is known as polymorphism

Ex:- Human face with different feelings

wax or mud with different shapes.

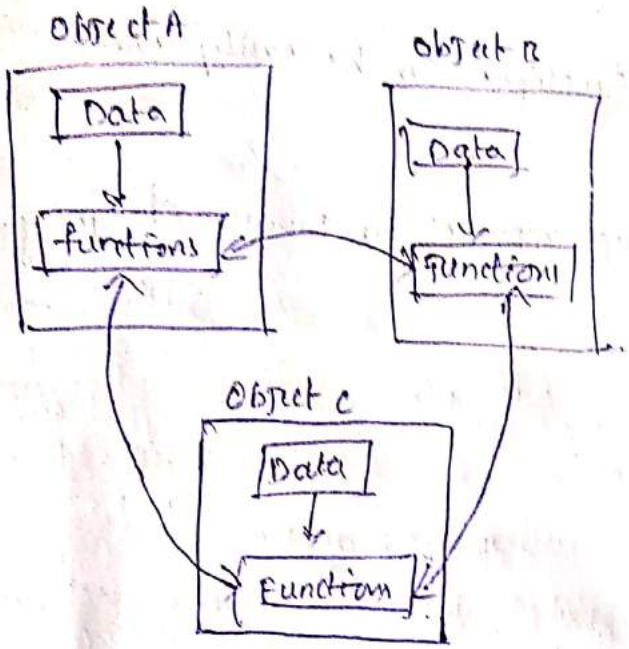
Abstraction:- Hiding internal details and showing functionality is known as Abstraction

for Ex:- Car driver only not know the internal functionality of Gear, Accelator, brake etc.

Encapsulation:- Binding (or) wrapping code and data together into single unit is called Encapsulation.

Ex:- capsule is wrapped with diffrent combination of medicines.

OOP paradigm:-



The major motivation factor in the invention of OO approach is to salvage some of the problems encountered in the procedural approach.

OOP treats data as a critical element in the program development and does not allow into flow freely around the system. It ties data more closely to the functions that operate on it and protects it from accidental modification from outside functions. OOP allows us to decompose the Prblm into number of entities called objects and then builds data and functions around these entities.

Features of OOP:-

→ Emphasis is on data rather than procedure.
→ Programs are divided into objects.
→ data structures are designed such that they characterize the objects
→ functions that operate on the data of an object (or) tied together in the data structure
→ data is hidden and cannot be accessed by external

functions

→ objects may communicate each other through functions.

→ New data and functions can be easily added whenever necessary.

→ Follows bottom up approach in program designing.

Unit-2 Classes and objects

Class:- A class is a way to bind the data and its associated functions together. It allows data and functions to be hidden if necessary from external use. When defining a class we are creating new abstract data type that can be treated like any other built in datatype.

Generally class specification has two parts

1. class declaration
2. class function declaration.

→ The class declaration describes the time and scope of its members. The function tells the behaviour of variables in a function.

General form of a class

```
class <name>
{
    private :                          member variables (&)
                                       data members.
      member variables ;
      member functions ;
    Public :
      member variables ;
      member functions
};
```

The class declaration is similar to structure declaration as shown above. The keyword class specifies that what follows is an abstract data of type name. The body of the class is enclosed within braces and terminated by a semicolon.

→ The class body contains the declaration of variables and functions. These variables and functions are collectively called class members.

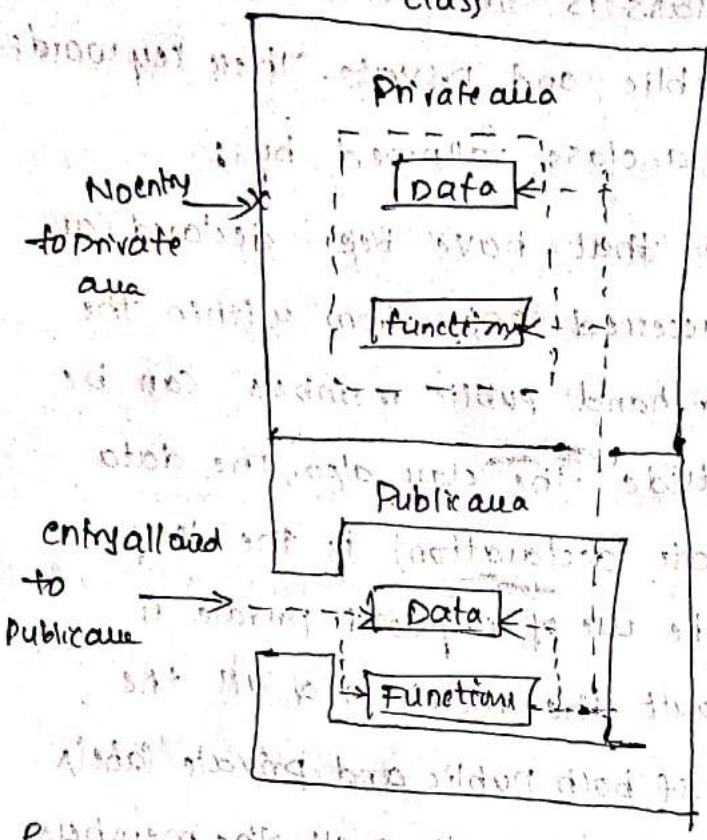
They are usually grouped under two sections namely public and private.

```
#include<iostream.h>
class abc
{
    private:
      int a;
      float b;
      char c;
    public:
     void get()
     {
       cout<<"enter int: ";
       cin>>a;
       cout<<"enter float:";
       cin>>b;
       cout<<"enter char:";
       cin>>c;
     }
     void put()
     {
      cout<<"Int= "<<a<<endl;
      cout<<"Float="<<b<<endl;
      cout<<"Char="<<c<<endl;
     }
};//end of class abc
int main()
{
   abc x;
   x.get();
   x.put();
   return 0;
}
```

enter int 3
enter float 4.5
enter char $
int=3
Float=4.5
Char=$

→ The scope of a class is grouped into two sections namely Public and Private. These keywords are placed inside a class followed by ;
the class members that have been declared as private can be accessed only from within the class. On the other hand public members can be accessed from outside the class also. The data hiding (using private declaration) is the key feature of OOP. The use of keyword private is optional. by default the members of all the class are private. If both public and private labels are missing within the class then all the members of a class are private then such a class is completely hidden from the outside world and does not serve any purpose.

→ The variables declared inside the class are known as data members and the functions are known as member functions. Only the member functions can have access to the private data members and private functions, public members (both data and functions) can be accessed from outside the class.



factorial of a number using class

```
#include<iostream.h>
class fact
{
    Private:
      int n;
      long f;
    Public:
      void input() {n;} fact() //default constructor
      {
        cout<<"enter n: ";
        cin >> n;
        f=1;
      }
      long fact()
      { int i;
        for(i=1;i<=n;i++)
        f=f*i;
        return f;
      }
};
```

enter n 5

factorial is 120.

```
int main()
{   fact b;
    b=input(); [→ dhinni rayakuda kuda when we use difficult computation
    long h= b.fac();
    printf(cout<<" factorial is "<<h;
    return 0;
}
```

2. Generate fibanacci series by using classes and objects

3.
```
1
1 2
1 2 3
1 2 3 4
```

4. $1! + 3! + \cdots n!$

5. $\frac{1}{1!} + \frac{2}{2!} + \frac{3}{3!} + \cdots \frac{n}{n!}$

6. printing numbers from n to 1

7. printing multiplication table.

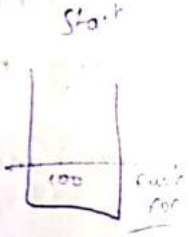
Inline functions:- one of the objectives of using functions in a program is to save some memory space which becomes appreciable when a function is likely to be called many times. However every time a function is called it takes a lot of extra time in executing a series of instructions for tasks such as jumping to the function, saving registers, pushing arguments into the stack and returning to the calling function. When a function is small a substanial percentage of execution time may be spent in such over heads.

In one of the solution to this problem is to use macro definition known as macros.

Preprocessor Macros are popular in c. The major drawback with macros is that they are not really functions and therefore the usual error checking does not occur during compilation.

Inorder to overcome these difficulties, c++ has Solution. has inline functions. An inline function is a function that is expanded inline when it is invoked that is the compiler replaces function call with the corresponding function code. The inline functions are defined as follows.

Ex:-
```
inline int sum(int a, int b, int c)
{ return(a+b+c);
}
void main()
{
  int x, y, z;
  cout << "enter x,y,z";
  cin >> x >> y >> z;
  int res = sum(x,y,z);
  cout << "sum =" << res;
}
```



→ It is easy to make a function inline all we need to do is to prefix the keyword inline to the function definition. all inline functions must be defined before they are called.

→ we must have to take some care before making a function inline function devinit as the function grows in size. At some point the overhead of the

function call becomes small compare to the execution of the function and benefits of inline function is lost. In such cases the use of normal functions will be more meaningful.

Some of the situations where inline expansions may not work are

1. for functions returning values if a loop, switch or goto exists
2. " " not returning " if a return statement exists
3. function contain static variables
4. If inline functions are recursive.

Inline expansion makes a program run faster. because the overhead of the function call and return is eliminated. However it makes the program to take up more memory because the statements that define the inline function are reproduced at each point where the function is called.

Static data Member functions

A data member of a class can be qualified as a static. The properties of a static member variables are similar to that of a static variable.

A static member variable has certain special characteristics.

They are.

1. It is Initialized to zero, when the first object of its class is created, No other Initialization is permitted.
2. only one copy of that member is created. for the entire class and shared it by all the objects of that class no matter how many objects are created.
3. It is visible only within the class but its lifetime is the entire program.

4. Static variables are normally used to maintain values common to the entire class.

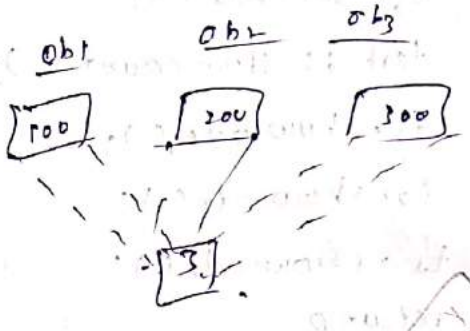
for example,
static data member can be used as a counter that records the occurences of all the objects.

Example:-

```
#include<iostream.h>
class item
{   static int count;
    int number;
  public:
    void getdata(int a)
    {   number=a;
        count++;
    }
    void getcount()
    {cout<<"count="<<count<<endl;
    };
};
int item::count;
int main()
{
    item a,b,c; //count=0
    a.getcount();
    b.getcount();
    c.getcount();

    a.getdata(100);
    b.getdata(200);
    c.getdata(300);
    cout<<"After reading data:"<<"\n";
```

```
a.getcount();
b.getcount();
c.getcount();
return 0;
}
```



o/p count=0
count=0
count=0
After reading data
count=3
Count=3
count=3

Static membu functions

Ex:-
```
class test
{
    int code;
    static int count;
  Public:
    void setcode()
    {
      code = ++count;
    }
    void showcode()
    { cout<<"object code: "<<code<<endl;
    }
    static void showcount()
    { cout<<"count: "<<count<<endl;
};  }
int test:: count;
int main()
{
    test t1,t2;
    t1.setcode();
    t2.setcode();
```

```
test :: showcount(); //accessing static function
test t3;
t3.setcode();
test :: showcount();
t1.showcode();
t2.showcode();
t3.showcode();
return 0;
}
```

count : 2
count : 3
object code : 1
object code : 2
object code : 3

Function overloading

Polymorphism

- Compile time Polymorphism
  - function ov
  - Constructor ov
  - operator ov
- Runtime Polymorphism

Ex:- In c++ polymorphism can be classified into two parts

1. compile-time polymorphism
2. Run-time polymorphism.

In compile-time polymorphism three types of polymorphism is allowed. They are

1. function overloading
2. Constructor overloading
3. operator overloading.

function overloading:- In c++ we can use same function name with different parameters. c++ compiler can recognize the overloading of a function.

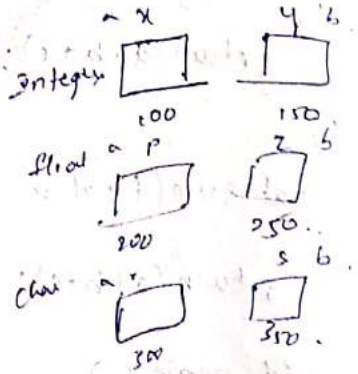
Ex:1

```
int sum (int a, int b, int c)
{
    return (a+b+c);
}
float sum(float a, float b, float c)
{
    return (a+b+c);
}
void main ( )
{
    int p, q, r;
    float x, y, z;
    cout << "enter 3 int values:";
    cin >> p >> q >> r;
    cout << "enter 3 float values:";
    cin >> x >> y >> z;
    cout << "sum of int value :" << sum(p,q,r) << endl;
}   cout << "sum of float values:" << sum(x,y,z) << endl;
```

Ex-2

```
void swap (int &a, int &b)
{
    int t;
    t = a;
    a = b;
    b = t;
}
void swap (float &a, float &b)
{
    int t;
    t = a;
    a = b;
    b = t;
}
void swap ( char &a, char &b )
{
    int t;
    t = a;
    a = b;
    b = t;
```

```
void main()
{
    int x,y;
    float p,q;
    char r,s;
    cout<<"enter 2 int:";
    cin>>x>>y;
    cout<<"enter 2 float:";
    cin>>p>>q;
    cout<<"enter 2 char:";
    cin>>r>>s;
    cout<<"Before swap:";
    cout<<"Integer:"<<x<<" "<<y<<endl;
    cout<<"float:"<<p<<" "<<q<<endl;
    cout<<"char:"<<r<<" "<<s<<endl;
    swap(x,y);
    swap(p,q);
    swap(r,s);
    cout<<"After swap:";
    cout<<"Integer:"<<x<<" "<<y<<endl;
    cout<<"float:"<<p<<" "<<q<<endl;
    cout<<"char:"<<r<<" "<<s<<endl;
}
```



enter 2 int 3 4

enter 2 float 3.7 4.5

enter 2 char r s

Overloading member functions:-

```
Class Box
{
  Private:
    double len, bre, dep;
  Public:
    void input();
    void input(double, double, double);
    void input(double);
```

```
    double vol( );
};
void Box :: input ( )
{
    cout << "enter len, bre, dep: ";
    cin >> len >> bre >> dep;
}
void BOX :: input (double l, double b, double d)
{
    len = l; bre = b; dep = d;
}
Void BOX :: input (double x)
{
    len = bre = dep = x;
}
double Box :: vol ( )
{
    return (len * bre * dep);
}
void main( )
{
    Box ob1, ob2, ob3;
    ob1.input( );
    ob2.input (4.2, 5.6, 7.7);
    ob3.input (5.5);
    Cout << "ob1 vol: " < ob1.vol ( ) << endl;
    Cout << "ob2 vol: " << ob2.voll ( ) << endl;
    cout << "ob3 vol: " << ob3.vol ( ) << endl;
}
```

enter len, bre, dep 2 3 4
ob1 vol : 24
ob2 vol : 181.104
ob3 vol : 166.375

double y;
cout << "enter y";
cin >> y;

Array of objects

An array may be any data type including struct. Similarly we can have arrays of variables that are of the type class. Such variables are called arrays of objects.

```
class emp
{
        char name[30];
        float age;
      public :
        void getdata();
        void putdata();
};
void emp :: getdata()
{
      cout<<"enter name";
      cin>>name;
      cout<<"enter age";
      cin>>age;
}
void emp:: putdata()
{
   cout<<"\n Name = "<<name;
   cout<<"\n age="<<age;
}
int main()
{
   emp m[3];
   cout<<" Details of 3 managers:";
   for(int i=0;i<3;i++)
   {
     cout<<" details of m"<<i+1<<endl;
     m[i].getdata();
   }
   cout<<"\n";
   for(i=0;i<3;i++)
   {
     cout<<" details of m"<<i+1<<endl;
     m[i].putdata();
   }
}
```

Objects as function Arguments

Like any other data type, an object may be used as a function argument. This can be done in two ways:

1. A copy of the entire object is passed to the function.
2. Only the address of the object is transferred to the function.

→ The first method is called pass by Value. Since a copy of the object is passed to the function any changes made to the object inside the function do not effect the object used to call the function

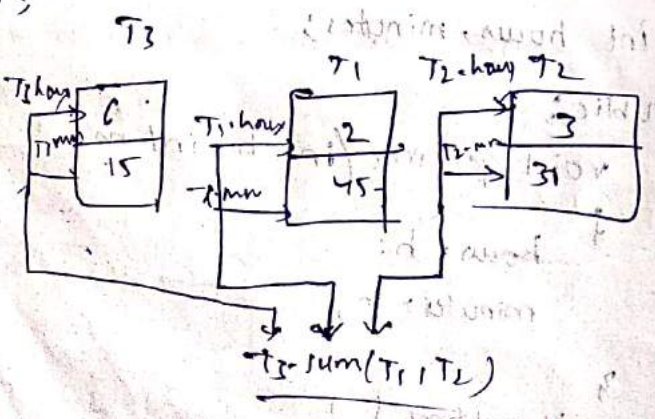
→ The second method is called pass by reference. When an address of the object is passed then the called function works directly on the actual object used in the call. This means that any changes made to the object inside the function will reflect in the actual object.

→ The pass by reference method is more efficient. Since it requires to pass only the address of the object and not the entire object.

```
Class time
{
    int hours, minutes;
  Public:
    void gettime (int h, int m)
    {
      hours = h;
      minutes = m;
    }
    void puttime ( )
    {
      cout<<hours<<"hours and ";
```

```
    cout << minutes << "minutes" << endl;
}
void sum (time, time);
};
void time :: sum (time t1, time t2)
{
    minutes = t1.minutes + t2.minutes;
    hours = minutes/60;
    minutes = minutes % 60;
    hours = hours + t1.hours + t2.hours;
}
int main()
{
    time T1, T2, T3;
    T1.gettime (2,45);
    T2.gettime (3,30);
    T3.sum (T1, T2);
    cout << "T1 = ";
    T1.Puttime();
    cout << "T2 = ";
    T2.Puttime();
    cout << "T3 = ";
    T3.Puttime();
    return 0;
}
```

T1 = 2 hour and 45 min
T2 = 3 and 30 min
T3 = 6 and 15 min



Local classes:- classes can be defined and used inside a function or a block. Such classes are called local classes.

```
void test (int a)
{
  ---------
  class student
  {
    ---------
    ---------
  };
  ---------
  ---------
  student s1(a);
}
```

Local classes can use global variables declared above the function and static variables are declared inside the function but cannot use automatic local variables. The Global variables should be used with scope operator (::)

There are some restrictions in constructing local classes. They cannot have static data members and static member functions.

Ex:-
```
void fun( )
{
  class test
  {
    public:
      void method ( )
      {
        cout<<"local class method"<<endl;
      }
  };
  test t;
  t.method ( );
}
int main( )
{ fun( );
  return 0;
```

Empty classes:- empty classes come from time to time in C++. C++ requires empty classes to have non zero size to ensure object identity. for Instance an array of empty class below has to have non zero size because each object Identified by the array subscript must be unique.

```
class EX
{

};
int main( )
{
  Ex a;
  cout<<"size of object EX is:"<<sizeof(a)<<endl;
  return 0;
}
```

Op: Size of object EX is 1.

In the above example Ex is a class having no data members and member functions called as empty class Size of an empty class is 1 but not zero.

Nested classes

A Nested class is a class which is declared in another enclosing class. A nested class is a member and as Such as the Same access rights as any other member. The members of an enclosing class have no special access to the members of a nested class. By usual access rules Shall be obeyed.

```
class A
{ public:
    class B
    { private:
        int num;
      public:
        void getdata(int n);
        { num=n;
        }
        void putdata( )
        { cout<< "the no is"<<num;
        }
    }; //end of class B
}; //end of class A
int main( )
{ cout<<"Nested classes in c++:";
  A::B obj;
  obj.getdata(9);
  obj.putdata( );
  return 0;
}
```

Nested class in c++:

return by reference

A c++ program can made easier to read and maintain by using references rather than pointers. A c++ function can return reference in a similar way as it returns a pointer.

→ when a function returns a reference it returns an implicit pointer to its return value. This way a function can be used on the left side of an assignment statement.

```
#include <iostream.h>
#include <time.h>
using namespace std;
double vals[] = {10.1, 12.6, 33.1, 24.1, 50.0};
double &setvalue(int i)
{ return vals[i];
}
int main()
{ int i;
cout<<" value before change:"<<endl;
for(i=0; i<5; i++)
{ cout<<"vals["<<i<<"]=";
  cout<<vals[i]<<endl;
}
setvalue(1) = 20.23;
setvalue(3) = 70.8;
Cout<<" values after change:"<<endl;
for(i=0; i<5; i++)
{ cout<<" vals["<<i<<"]=";
; cout<<vals[i]<<endl;
}
```

** Friend functions: Non member functions of a class cannot have access to the private members of a class. But there is a situation where we would like two classes to share a particular function, then non member function which is friendly to both classes can share private data.

Syntax is

```
class <name>
{
    private :
    - - -
    - - -
    public :
    - - -
    friend returntype functionName(parameter);
};
```

→ The function declared should be preceded by the keyword friend.

→ The definition of the function defined elsewhere in the program like a normal c++ function. The function definition does not use either keyword friend or the scope operator.

→ A function can be declared as a friend to any number of classes. A friend function although not a member function has full access rights to the private members of the class.

Characteristics of friend function

→ It is not in the scope of the class to which it has been declared as a friend.

→ Since it is not in the scope of the class it cannot be called using the object of that class

→ It can be Invoked like a normal function without any object

→ unlike member functions it cannot access the member names directly and has to use an Object name and dot membership operator and each member name.

→ It can be declared either in public or Private region of

without effecting its meaning

-usually it has the objects like arguments

```
class sample
{
    int a, b;
    public:
    void setvalue()
    { a=25
      b=40;
    }
    friend float mean(sample s);
};
float mean(sample s)
{ return float(s.a+s.b)/20;
}
int main()
{ sample x;
  x.setvalue();
  cout<<"mean value="<<mean(x)<<endl;
}
```

```
class ABC; //forward declaration
class XYZ
{
    int x;
    public:
    void setvalue(int i)
    {
      x=i;
    }
    friend void max(XYZ, ABC);
};
```

```
class ABC
{   int a;
    Public:
    void setvalue (int i)
    { a=i;
    }
   friend void max(XYZ, ABC);
};
void max( XYZ m, ABC n)
{
    if (m.x > n.a)
      cout << m.x << "is max value".
    else
     cout<<n.a<<" is max value";
}
int main()
{
    ABC abc;
    abc.setvalue (10);
    XYZ xyz;
    xyz.setvalue (20);
    max(xyz, abc);
    return 0;
}
```

Friend class:- A friend class can access private and protected members of other class in which it is declared as friend. It is sometimes useful to allow a particular class to access private members of other class.

Ex:- A linked list class may be allowed to access private members of node as shown below

```
Class Node
{
    Private:
      int num;
      node *next;
   Publil:
    friend class linkedlist;
};
class linkedlist
{  Public:
    Void create ( );
    Void input ( );
    void delt );
    Void disp ( );
};
```

3, Constructors

[Explicit call → manampilavali
Implicit call → automatic calling.
(constructor works only for initialization but not for logic operation)]

Contructors:- A constructor is a Special member function whose task is to Initialize the object of its class. It is Special because its name is the Same as the class name. The constructor is invoked whenever an object of its associated class is created. It is called constructor because it constructs the value of data members of the class.

*Note:- constructor does not contain any return type. Function is having explicit call and constructor having implicit call

```
#include<iostream.h>
class Box
{ Private:
  double len, br, he;
  Public:
  Box( ) // Default constructor
  { cout<<"enter len br he: ";
    cin>>len>>br>>he;
  }
  double vol( )
  { return len * br * he;
  }
};
void main( )
{
  Box ob;
  cout<<"volume="<<ob.vol( );
}
```

enter len, br, dep
20 12 13
volume 3120

→ Constructor functions have Some Special Characteristics. They are.

1, They should be declared in Public Section.

2 They are Invoked automatically when the objects are created.

This is called dynamic invocation. That means without calling constructor will be called automatically Immediately after the object is created

→ They don't have return type not even void. Therefore They cannot return any value.

→ Constructor are used only for initialization which is called dynamic initialization. They cannot be used for other logical purposes except initialization.

→ They cannot be inherited. Though derived class can call the base class constructor.

→ like other c++ functions, They can have default arguments → Constructor can be Parameterized.

→ constructors cannot be virtual

→ they cannot refer to their addresses.

→ An object with a constructor cannot be used as a member of a union.

→ A constructor without any arguments is called default constructor.

J. Constructors can be overloaded same as function overloading.

Parameterised constructor

```
class BOX
{
    Private:
    double len, bre, ht;
    Public:
```

```
Box (double l, double b, double h)
{
    len = l;
    bre = b;
    he = h;
}
double vol()
{
    return len*bre*he;
}
void main()
{
    Box obj(2.8, 3.3, 6.6); //Implicit call
    Box ob2;
    ob1 = Box(5.5, 6.2, 9.1); //explicit call
    cout << "vol of ob1 = " << ob1.vol() << endl;
    cout << "vol of ob2 = " << ob2.vol() << endl;
}
```

Function definition → body variables

Value will invoked and initialization

60.984

248.93

Default arguments in a constructors

```
class BOX
{
    private:
        double len, bre, he;
    Public:
        BOX(double l = 5.5, double b = 6.6, double h = 9.2);
        double vol();
};

BOX :: Box(double l, double b, double h)
{
    len = l;
    bre = b;
    he = h;
}
double box :: vol()
{
    return len*bre*he;
```

(2) 261.36

(1) 347.76

```
}
void main()
{
    Box Obj(5.5, 8.2, 7.6);
    Box Obj2;
    cout << "volume is " << Obj.doublevol();
    cout << "volume is " << Obj2.doublevol();
}
```

Overloading of constructors

→ we can use overloading of a constructor which is similar to function overloading but constructors can be used only for Initialization.

```
class BOX
{
    private:
        double len, bre, hi;
    Public:
        BOX() // default constructor.
        Box(double, double, double);
        Box(double);
        [Box(Box &); // copy constructor] → optional
};
Box :: Box()
{
    len = 5.6;
    bre = 3.2;
    hi = 6.8;
}
Box :: Box(double l, double b, double h)
{
    len = l; bre = b; hi = h;
}
```

```
Box :: Box (double c)
{  len = bre = hi = c;
}
double Box :: vol ( )
{
    return (len*bre*hi);
}
void main()
{  Box ob1 ob2(6.2, 5.3, 3.7), ob3(8.2);
cout << " vol of ob1 =" << ob1.vol() << endl;   121.856
cout << "vol of ob2 =" << ob2.vol() << endl;    253.022
cout << "vol of ob3 =" << ob3.vol() << endl;    551.368.
}
```

Copy Constructor :-

A constructor is used to initialize an object from another object. It is also called as object cloning. This construct is also called as cloning constructor.

Ex:- `Box (Box &ob); // constructor`

```
Box ob2(ob1);   // object cloning or
(or)            // copying ob1 into ob2
Box ob2 = ob1;  // ob2 becomes duplicate
                // copy of ob1
```

```
class Box
{   private:
    double len, bre, hi;
    public:
    Box ( )
    {
        len = 7.2;
        bre = 6.3;
    } hi = 5.2;
```

```
Box (Box &ob)
{
    len = ob.len;
    bre = ob.bre;
    hi = ob.hi;
}
double vol()
{
    return (len * bre * hi);
}
void main()
{ Box ob1; // Default constructor called.
  Box ob2(ob1); // object cloning
                // Box ob2 = ob1; -> // alternative explicit call.
  cout << "vol of ob1=" << ob1.vol() << endl; 254.592
  cout << "vol of ob2=" << ob2.vol(); 254.592
}
```

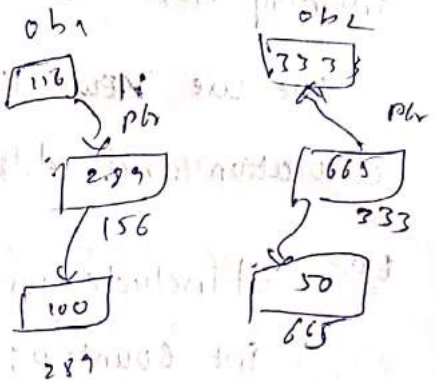
Dynamic Constructor

The constructor can also be used to allocate the memory while creating objects. This will enable the System to allocate the right amount of memory for each object. when the objects are not in the same size, thus resulting in saving of memory

Allocating of memory to the objects at the time of their construction is known as dynamic construction of objects. The memory is allocated with the help of new operator.

Ex:- class dyncon

```
{
    int *ptr;
  Public:
    dyncon ()
    {
      ptr = new int;
      *ptr = 100;
    }
    dyncon (int v)
    { ptr = new int;
      *ptr = v;
    }
    int display ()
    { return *ptr;
    }
};
Void main ()
{
    dyncon ob1, ob2(50);
    cout<<"In the value of ob1 ptr is: ";
    cout<<ob1.display ();
    cout<<"In the value of ob2 ptr is: ";
    cout<< ob2.display ();
}
```



Destructor and dynamic destructor

Destructor is used to destroy the objects have been created by a constructor. like constructor the destructor is a member function whose name is same as the class name but it is preceded by a 'tilde' operator.

Ex:- ~Box ()

→ A destructor never takes any arguments nor does it return any value. It will invoke implicitly by the

compiler upon the exit from the program. To clean up the storage, the destructor will be used. It is a good practice to declare the destructors in a program since it releases memory space for future use.

We use new keyword for dynamic memory allocation and delete for freeing the memory.

Ex:-
```
#Include<iostream.h>
int Count=0;
Class alpha
{
  public:
   alpha( )
   { count ++;
     cout<<"\n No. of objects created:"<<Count<<endl;
   }
   ~alpha( )
   { cout<<"\n No. of objects destroyed:"<<count;
     Count--;
   }
};
int main( )
{
  cout<<"\n\n Enter main\n";
  alpha a1, a2, a3, a4;
  { cout<<"\n\n Enter Block 1\n";
    alpha a5;
  }
  {
   cout<<"\n\n Enter Block 2\n";
    alpha a6;
  }
  cout<<"\n\n RE-Enter main\n";
  return 0;
}
```

Scanned by CamScanner

Dynamic constructor

```
class string
{   char *name;
    int length;
    public:
    string()
    {   length=0;
        name = new char[length+1];
    }
    string(char *s)
    {   length = strlen(s);
        name = new char[length+1];
    }
    void display()
    {   cout<<"Name = "<<name<<length;
};  }
    void string :: join(string a, string b)
    {   length = a.length + b.length;
        delete name;
        name = new char[length+1];
        strcpy(name, a.name);
        strcat(name, b.name);
    }
    void main()
    {   char *first = "Albin";
        string name1(first), name2("Ramesh"),
               name3("Tendulkar"), s1, s2;
        s1.join(name1, name2);
        s2.join(s1, name3);
        name1.display();
        name2.display();
        name3.display();
        s1.display();
        s2.display();
    }
```

Anonymous objects:-

Objects are created with names after declaring a class we can create an object with some name. so it is possible to declare objects without name such objects are known as Anonymous objects.

We already know how to invoke constructors and destructors. When constructors and destructors are invoked the data members of the class are initialized and destroyed respectively. Thus without object we can initialize and destroy the content of the class.

Ex:-

```
class xyz
{
    private :
      int x,y,z;
    public:
     xyz ()
     { cout<<"\n Inside constructor \n";
       x=5;
       y= 6;
       z=7;
       show ();
     }
     void show()
     { cout<<"x:"<<x<<endl;
       cout<<"y:"<<y<<endl;
     }  cout<<"z:"<<z<<endl;
     ~xyz ( )
      { cout<<"\n Inside destructor \n";
      }
};
int main ( )
{    xyz ();
}    return 0;
```

Operator overloading:- [only in c++]

It is one of the important features of c++ language. this enhances the extentability of c++. this concept is advantageous when built in operators meaning is enhanced. for example:- we can use + for adding two built in type variables. we can extend this to add two user defined data types.

This means that c++ has the ability to provide the operators ~~and~~ with a special meaning for datatype. The mechanism of giving such special meaning to an opuator is known as operator overloading.

→ we can overload all the c++ operators except a

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

Note:- when an operator is overloaded its original meaning must not be changed.

Syntax:-

```
returntype classname :: operator op(arg)
{
    body
}
```

→ operator functions must be either member functions or friend functions.

→ A friend function will have only one argument for unary operators and two for binary operators.

→ A member function will have no arguments for unary operators and only one for binary operators.

→ Arguments can be passed either value or by reference.

Some examples:

1. Vector operator + (vector)
2. vector operator - ( )
   // unary minus for Negation
3. friend vector operator + (vector, vector)
   // Binary + friend function.
4. friend vector - (vector)
   // unary minus friend function is used.
5. int operator == (vector)
   // Binary == is overloaded.

→ The process of overloading involves the following steps.

1. create a class that defines the data type that is to be used in the overloading operation.
2. Declare the operator function operator op( ) in the public part of the class. It may be either member function or friend function
3. Define the operator function to implement the required operations.

Overloading Unary minus (-) operator

```
class unary
{
    private:
        int x;
        float y;
```

```
    double z;
Public:
    unary ()
    { cout << " enter int, float, double!"
      cin >> x >> y >> z;
    }
    Void show ()
    { cout << "Int x:" << x << endl;
      cout << "Float y:" << y << endl;
      cout << "double z:" << z << endl;
    }
    void operator -()
    {   x = -x;
        y = -y;
        z = -z;
    }
};
int main()
{
    unary m;
    cout << "\n Before overload \n";
    m.show();
    -m;
    cout << " \n After overload \n";
    m.show();
}
```

Implicit argument is available

explicit argument we have to pass the parameter

unary overloading using friend function we have to pass the parameter in unary overloading also

```
friend void operator -(unary);
void operator -(unary n)
{   n.x = -n.x;
    n.y = -n.y;
    n.z = -n.z;
}
```

Implicit

description:- In the above program, we used unary minus as a member function in order to overload the operator (-) for negating an object.

→ if the member function is used when we call the member (-) followed by an object. This object is implicitly available for the member function (-)

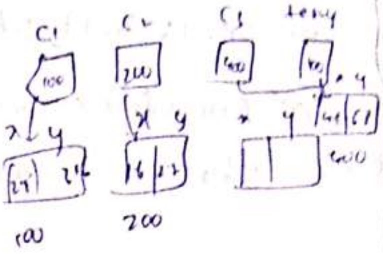
When we are using the friend function for the same, we need to send the object as explicit parameter

Overloading binary plus (+)

```
class complex
{
    Private:
     float x,y;
    public:
     complex() { }
     complex(float real, float imag)
     {
         x= real;
         y= imag;
     }
     complex operator + (complex ) friend complex operator + (
     void display();
};
complex complex:: operator + (complex c)
{
   complex temp;
   temp.x = x+c.x;
   temp.y = y+c.y;
   return (temp);
}
void complex:: display()
{
   cout<<x<<"i"<<" + j "<<y<<endl;
}
int main()
{
   complex c1, c2, c3;
   c1 = complex(2.5, 3.6);
   c2 = complex(1.6, 2.7);
   c3 = c1 + c2 ; (I thada + nichmi + function call anuthudh)
```

complex operator + (complex c)
{ complex temp;
temp.x = x1.x + x2.x)
temp.y = x1.y + x2.y;
return (temp);
}

```
cout<<"c1=";
c1.display();
cout<<"c2=";
c2.display();
cout<<"c3=";
c3.display();
return 0;
}
```



Implicitly the as need to be Parameter; operator to left side Implicitly available Right side variable explicitly available so we have to send as a Parameter

Inside temp we have maintained value c1 then add c2 to the ...

III. this Pointer [usage of this pointer, significance]

(currently involving object pointer.)

C++ uses a unique keyword this to represent an object that invokes a member function. this is a pointer that points to the object for which this function was called.

```
class Box
{
    Private:
      double l, b, h;
   Public:
     Box()
     { cout<<"Enter l, b, h";
       cin>>l>>b>>h;
     }
     Box(double l, double b, double h)
     { this->l = l;
       this->b = b;
       this->h = h;
     }
     double vol()
     { return l*b*h;
     }
};
void main()
{
```

```
Box ob1, ob2(5,6,7), ob3(9,8,9);
cout << "ob1 vol=" << ob1.vol();
cout << "ob2 vol=" << ob2.vol();
}
```

overloading new and delete operators

```
class student
{
    string name;
    int age;
Public:
    student()
    { cout << "constructor called \n";
    }
    student(string name, int age)
    { this->name = nam;
      this->age = age;
    }
    Void display()
    { cout << "Name:" << name << endl;
    } cout << "age:" << age << endl;
    void * operator new(size_t size)
    {
    cout << "overloading new operator with size:" <<
                                    size << endl;
    void *p = :: new student();
    // void *p = malloc(size); will also work here
    return p;
    }
    Void operator delete(void *p)
    {
```

```
    cout << "overloading delete operator" << endl;
    free(p);
  }
};
int main()
{ Student *p = new Student ("XYZ", 20);
  p->display();
  delete p;
}
```

Overloading unary -- ++ operator (using member function as well as friend function)

Program:-

```
class uppumm
{
  private:
    int a;
    long b;
    char c;
    double d;
  public:
    uppumm()
    { a=25;
      b=156;
      c='A';
      d=255.65;
    }
    void show()
  { cout<<"Int a = "<<a<<endl;
    cout<<"long b:"<<b<<endl;
    cout<<"Char c="<<c<<endl;
    cout<<"double d:"<<d<<endl;
  }
```

```
    void operator --( ); friend void operator --(UPPUMM &);
    void operator ++ ( ); friend void operator ++(UPPUMM &);
};
void UPPUMM :: operator ++( ) void operator ++(UPPUMM &x)
{                                  {
  a=a+25;                            x.a = x.a+25;
  b= b+36;                           x.b = x.b+36;
  c= c+8;                            x.c = x.c+8;
  d= d+65.26;                        x.d = x.d+65.26;
}                                  }
void UPPUMM :: operator --( )      void operator --(UPPUMM &x)
{ a= a-13;                         {
  b= b-26;                           x.a = x.a-13;
  c= c-3;                            x.b = x.b-26;
  d = d-35.62;                       x.c = x.c-3;
}                                    x.d = x.d-35.62;
void main( )                       }
{
    UPPUMM obj;
    cout<<"In Before overload ++ \n";
    obj.show ( );
    ++obj;
    cout<<"In After overload \n";
    obj.show( );
    Cout<<"In Before overload -- \n";
    obj.show( );
    --obj;
    cout<<"In After overload \n";
    obj.show( );
}
```

→ overloading binary ++ increment according to the Lebanacci number

⇒ by using binary + add two corresponding elements in two objects of same class

⇒ Overloading new and delete operator in C++

The new and delete operator can also be overloaded like other properties in C++.

New and delete operator can also be overloaded globally or they can be overloaded for specific class.

if these operator are overloaded using member function for a class it means that these operator are overloaded only for that specific class.

if overloading is done outside a class (for eg:- it is not a member function of a class) the overloaded new and delete will be called any time, we make use of these operator within classes outside class. This is also called as Global overloading.

Syntax for overloading new operator:-

⇒ void *operator new(size_t, size);

description:-

The overloaded new operator receives size of type (size_t) is specified the number of bytes of memory to be allocated.

The return type of the overloaded new must be void *

The overloaded function returns a Pointer to the beginning of the block of memory allocated.

Syntax for overloading delete operator:-

⇒ void operator delete (void *)

The function receives a parameter of type void * which has to be deleted. Function should not return anything.

Note:- Both overloaded new and delete operators are static members by default, Therefore, They don't have access to this pointer.

Type Conversion in c++:-

A type cast or type conversion from one type to another. they are two types of type conversions

1. implicit type conversion
2. explicit type conversion.

Implicit type conversion:-

It's automatic type conversion that means one datatype is coverted into another datatype by compiler itself done its duty without any request from the user.

Generally, type conversions are done from lower Cast to upper cast

for eg:- Type casting is done in the following order

↓ bool → char → short int → int → unsigned int → long → unsigned → long long → float → double → long double.

one byte

Ex:- Implicit type cast

```
int a=10;
float b=25.6;
double c=a+b;
```

o/p → 35.6

In the above example, the variable a is an integer and b is a float variable. if we are adding these two

Variables. a is converted into float first and the resultant is converted into double next implicitly by the compiler by using "Automatic type conversion technique."

Note:- only lower cast can be converted as upper cast but vice versa is not true.

explicit type conversion:-

It's user defined type casting. Here the user can be type cast the result to be make it of the particular datatype.

Ex:- $1 + \frac{1}{1!} + \frac{2}{2!} + \cdots + \frac{n}{n!}$

```
Void main()
{  int n;
   float factsum(int);
   float x;
   x = factsum (5);
   cout<<x;
}
float factsum(int m)
{  int i, j;
   float s=0;
   for(i=1;i<=m;i++)
   {  s+= (float)i/ fact(i);
   }
   return s;
}
long fact(int n)
{  int i, long f=1;
   for(i=n;i>1;i--)
      f=f*i;
   return f;
}
```