

Unit 4:

More Extension to C in C++ :

OBJECT ORIENTED PROGRAMMING
FACILITIES

Outline

1. Scope of Class and Scope Resolution Operator
2. Member Function of a Class
 1. Declaration inside the class
 2. Declaration outside the class
3. friend function
4. this Keyword
5. constructors and Destructors
6. friend class
7. Error handling (exception)

Local Class and Global Class

- ❖ Definition point of the class state the scope of the class: Local or Global
- ❖ If class is defined within the function then scope is limited to that function only. No other function is allowed to create the object of that class. (Local)
- ❖ If class is defined independent of any function then it will be accessible to all other functions defined below that class. (Global)
- ❖ If Name of the Local Class and Global class is same in the same program then Global class object can be created by using scope resolution operator (::).

Syntax - `:: class_name object_name;`

- ❖ Complete example shown in **Program 1**

Scope of Class :

```
#include<iostream>
int main()
{
```

```
class xyz
{
int x;
public:
void get()
{
cin>>x;
}
void put()
{
cout<<x;
}
};
```

```
xyz x;
x.get();
x.put();
}
```

Local Class: declared in
main function

Only
allowed in
main
function
since class
declaration
is in scope
of main
function
only. Kind
of a private
class

```
void test()
{
xyz x;
x.get();
x.put();
}
```

Not allowed
since class
declaration
is in scope
of main
function
only.

Global Class: declared
independently

```
class XYZ
{
};

int ABC()
{
XYZ x1;
}

void fun1()
{
XYZ x2;
}

int main()
{
XYZ x3;
}
```

Global Class: Object
declaration is allowed in
all the functions

Example 1: Scope of the class

Problem Statement:

- ❖ Write a program to calculate total volume of different shapes such as **cube, sphere and cylinder**. Create a class **"VOLUME"** with the data member such as **Volume of cube, Volume of sphere and Volume of cylinder**. Write appropriate member functions to calculate volume of the shapes mentioned in the class. Implement this class using main function.
- ❖ Later on, we need to add one more shape and calculate the volume and add it to volume of the other shapes of the class. What is the solution?
- ❖ Instead of making changes in the existing class, we can declare Local class "volume" in main function and it will solve the purpose.

Program 1:

```
#include<iostream>
```

```
class volume
```

```
{
```

```
float v;
```

```
public:
```

```
float vol(float s)
```

```
{v = s*s*s;
```

```
return v;}
```

```
float volS(float r)
```

```
{ v = 4.0/3.0*3.14*r*r*r;
```

```
return v; }
```

```
float vol(float r, float h)
```

```
{ v= 3.14*r*r*h;
```

```
return v; }
```

```
};
```

Global Class:
declared
independently

Local Class: declared in
main function with same
name

Example of function
overloading

Use of scope resolution operator
to declare object of Global Class
with same class name

```
int main()
```

```
{
```

```
class volume
```

```
{
```

```
int v;
```

```
public:
```

```
float vol(float r, float h)
```

```
{
```

```
v= 3.14 * r*r* (h/3);
```

```
return v;
```

```
}
```

```
};
```

```
volume V; // Object of Local class
```

```
float addV;
```

```
:: volume V1; // Object of Global class
```

```
addV = V.vol(3.0,6.0) + V1.vol(4.0) + V1.volS(6.0) +  
V1.vol(3.0,8.0);
```

```
Cout<<"addition of volume is"<<addV;
```

```
}
```

Scope resolution operator (::)

- ❖ It is also used to define the scope of local and global variables
 - ❑ If the global variable name is same as local variable name, the scope resolution operator will be used to call the global variable.
- ❖ It is used to define a member function outside the class
- ❖ It is also used with static data members

Examples 1:

Scope resolution operator (::) with global and local variable

```
//Example 1:
#include <iostream>
using namespace std;
char a = 'm';
int main()
{
char a = 's';

cout << "\nThe local variable : " <<
a;
cout << "\nThe global variable : " <<
::a;
return 0;
}
```

```
//Example 2:
#include <iostream>
using namespace std;
int my_var = 0;
int main()
{
int my_var = 0;
::my_var = 1; // set global my_var to 1
my_var = 2;   // set local my_var to
2
cout << ::my_var << " " << my_var;
return 0;
}
```


Example 2:

Scope resolution operator (::) Defining member function outside the class

```
class Student
{
    char name[20];
    float Percentage;
    int meritNo;
public:
    void readData()
    {
        cout<<"Enter Name of the
Student :";
        cin>>name;
        cout<<"Enter Percentage of the
Student :";
        cin>>Percentage;
    }
    void sorting(Student s[],int n);
    void assignMerit(Student s[], int n);
```

```
void printMerit()
{
    cout<<"Name of the student
:"<<name<<endl;
    cout<<"Percentage is
:"<<Percentage<<endl;
    cout<<"Merit No is :"<<meritNo<<endl;
}

void Student :: assignMerit(Student s[],int n)
{
    for(int i = 0;i<n;i++)
    {
        s[i].meritNo=i+1;
    }
}
```

```
void Student :: sorting(Student s[], int
n)
{
    Student s1;
    for(int i = 0;i<n;i++)
    {
        for(int j=i+1;j<n;j++)
        {
            if(s[i].Percentage < s[j].Percentage)
            {
                s1=s[i];
                s[i]=s[j];
                s[j]=s1;
            }
        }
    }
}
```

Example 2: main() function

Scope resolution operator (::) Defining member function outside the class

```
int main()
{
    Student s[5];
    cout<<"Enter data of the student : "<<endl;
    for(int i = 0;i<3;i++)
    {
        s[i].readData();
    }
    s[0].sorting(s,5);
    s[0].assignMerit(s,3);
    for(int i = 0;i<5;i++)
    {
        s[i].printMerit();
    }
}
```

Topic 2:

FRIEND FUNCTION AND FRIEND CLASS

Friend function:

Scenario : Use of objects of two classes together in the main function using friend function

Class ABC

```
{  
private: data members  
private: member functions  
public: member functions  
friend void Fun1(ABC,XYZ);  
}
```

Class XYZ

```
{  
private: data members  
private: member functions  
public: member functions  
friend void Fun1(ABC,XYZ);  
}
```

Declaring function as
a friend to class A
and class B : function
signature

Defining function
independently : it is
not a member of
class

int main()

```
{  
ABC a1;  
XYZ x1;  
Fun1(a1,x1); // calling of a friend function  
}
```

void Fun1(ABC a, XYZ x)

```
{  
  
}
```

friend function:

friend function of a class – **special function for specific use**

- ❖ Defined outside that class's scope – body of the function declare outside of the class.
 - ❖ Not a member function of that class
- ❖ Yet has the right to access the **private** (and public) members of that class
- ❖ Standalone functions or entire class may be declared to be friends of a class
- ❖ It is possible to specify overloaded functions as friends of a class
 - ❖ Each overloaded function intended to be a friend must be explicitly declared as a friend of the class
- ❖ Often appropriate when a member function cannot be used for certain operations

Example 1: friend function - Syntax

```
#include<iostream.h>

class XYZ; //prototype of class

class ABC
{
int a;
public :
void reada(int a1) { a = a1;}
friend void sum(ABC A, XYZ X);
};
```

Declaring function
sum as a friend to
class XYZ and ABC :
function signature

```
class XYZ
{
int x;
public :
void readx(int x1) { x = x1;}
friend void sum(ABC A, XYZ X);
};

void sum(ABC A, XYZ X)
{
int s= A.a +X.x
cout<<s;
}
```

Defining function
independently : it
is not a member
of class

```
int main()
{
ABC A1; // object of class ABC
A1.reada(10);
XYZ X1; //object of class XYZ
X1.readx(20);
sum(A1,X1);
}
```

Calling of a
friend
function
without object

Example2: friend function

```
#include<iostream.h>

class Complex2;

class Complex1
{
float real;
float img;
public :
void readNumber(fload r, float i)
{ real = r;  ing = i; }
Void printNumber() {
Cout<<real<<" + i "<<img;
}

friend Complex1 sum(Complex1 C1,
Complex2 C2);
};
```

```
class Complex2
{
float real;
float img;
public :
void readNumber(fload r, float i)
{ real = r;  ing = i; }

friend Complex1 sum(Complex1 C1,
Complex2 C2);
};

Complex1 sum(Complex1 C1, Complex2
C2)
{   Complex1 c3;
C3.real = C1.real + C2.real;
C3.img = C1.img + C2.img;
Return C3;
}
```

// Addition of two complex numbers

```
int main()
{
Complex1 A1,A2; // objects of class
A1.readNumber(4.5,3.4);

Complex2 X1; //object of class
X1.readNumber(2.4,5.2);

A2 = sum(A1,X1);
A2.printNumber();
}
```

Example3 : friend function

Statement:

Write a program to calculate the results of a student by combining internal and external marks out of 100. Write two separate classes as “Internal” and “External” with following data members common to both the classes. (out of 50 marks for each class)

- Marks of Math
- Marks of OPP
- Marks of SE
- Marks of DS

Use member function to input the marks for each class. Define friend function to calculate the final result by combining internal and external marks and print the total marks subject wise.

Program: Example 3

Class Internal

```
{
int OOP, Math, SE, DS;
public:
void readMarks()
{
cout << "Enter internal marks of four
subjects";
cin>> OOP>>Math>>SE>>DS;
}
friend void Sum(Internal, External);
};
```

Class External

```
{
int OOP, Math, SE, DS;
public:
void readMarks()
{
cout << "Enter external marks of four
subjects";
cin>> OOP>>Math>>SE>>DS;
}
friend void Sum(Internal, External);
};
```

void Sum(Internal I1, External E1)

```
{
}

int main()
{
Internal I; I.readMarks();
External E; E.readMarks();
Sum(I,E); // calling of a function
}
```

Homework: You have to write complete function Sum() as we have discussed in the class

friend class:

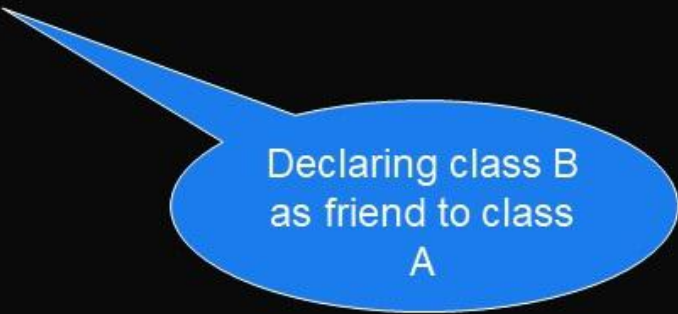
- ❖ Friendship is granted, not taken
 - ❖ For class B to be a friend of class A, **class A must explicitly declare that class B is its friend**
- ❖ Friendship relation is neither **symmetric** nor **transitive**
 - ❖ If class A is a friend of class B, and class B is a friend of class C, you cannot infer that class B is a friend of class A, that class C is a friend of class B, or that class A is a friend of class C
- ❖ To access private members of the class A in friend class B, an object of class A is required in function of friend class B.

friend class : - Syntax

```
class B;  
class A  
{  
int a;  
// class B is a friend class of class A
```

```
friend class B;  
}
```

```
class B  
{ ... ..  
int x;  
public:  
void getdata(A obj) { cin>>x;  
cin>>obj.a; }  
}
```



Declaring class B
as friend to class
A

- When a class is made a friend class, all the member functions of that class becomes friend functions.
- In this program, all member functions of class B will be friend functions of class A. Thus, any member function of class B can access the private and protected data of class A.
- But, member functions of class A cannot access the data of class B.
- Remember, friend relation in C++ is only granted, not taken.

Example: reading and printing data member

```
#include<iostream.h>

class XYZ; //prototype of class

class ABC
{
    int a;
    public :
    void reada (int a1) { a = a1;}
    void printa() { cout<< a; }
    friend class XYZ;
};
```

Declaring class
XYZ as friend class
to ABC

```
class XYZ
{
    int x;
    public :
    void readx (int x1) { x = x1;}
    void printx (ABC A)
    {
        cout<< x;
        cout<<A.a;
    }
};
```

Function printx() –
accessing private data
member of class ABC
with its object

```
int main()
{
    ABC A1; // object of class ABC
    A1.reada(10);
    XYZ X1; //object of class XYZ
    X1.readx(20);
    X1.printx(A1);
}
```

X1 object of XYZ
class will print the
data of class ABC
(data of object A1)

Example 2 : friend class

Problem statement : Write a program to demonstrate friend class behaviour. Friend class functions will access the data members of the other class.

A. Write a class "Employee" to read and print information of an employee with following details

Data members: Name of the employee, Id of the employee, Department of the employee, monthly salary

B. Write another class "Sales" that read and print employee details as follows

Sales in month, Incentives acquired – 10 % of the monthly salary if units sold between 50 to 100, 20 % of monthly salary if units sold is in between 100 and 150, and 30 % of monthly salary if more than 150 units sold.

Later on company has decided to include performance indicator depending on total sales made in a month as Excellent, Good, Satisfactory or Poor. Add one more function in sales class to assign performance indicators to the employee.

Use following conditions:

1. Sales above 150 – Excellent
2. Sales between 101 to 150 – Good
3. Sales between 50 to 100 – Satisfactory
4. Sales below 50 - Poor

Topic 3:

CONSTRUCTORS AND DESTRUCTORS

Why constructors?

- ❖ When an object is declared and if it is not initialized, it contains the **garbage value**.
- ❖ The programmer needs to **initialize objects** with appropriate values.
- ❖ This can be done through the **public member functions**.
- ❖ In the first step the object is created and in the second step the data members are initialized through public member functions.
- ❖ It could be better if the initialization is done at the time the object is created or declared.
- ❖ This can avoid calling the public member function to initialize the data members. This initialization is possible with **constructors**.
- ❖ **Therefore constructors are used to initialize object without member function**

What is Constructors?

- ❖ A constructor is a **special member function** whose task is to **initialize the objects** of its class.
- ❖ It is special because its name is same as the class name.
- ❖ The constructor is **invoked (implicitly)** whenever an object of its associated class is created.
- ❖ It is called constructor because it constructs the values of data members of the class.

obj

100

50

Example 1: Constructor syntax

*****Syntax 1:*****

```
class add
{
    int m, n ;
```

```
public :
```

```
    add ()
```

```
    {
```

```
        m=100;
```

```
        n=50
```

```
    }
```

```
--
```

```
};
```

Constructor
defined inside
the class

*****Syntax 2*****

```
class add
```

```
{
```

```
    int m, n ;
```

```
public :
```

```
    add () ; //constructor
```

```
    -----
```

```
};
```

```
add :: add ()
```

```
{
```

```
    m = 0; n = 0;
```

```
}
```

Constructor
defined out
side the class

➤ When a class contains a constructor, it is guaranteed that an object created by the class will be initialized automatically.

add obj; // object created and initialized

➤ Not only creates the object A of type add but also initializes its data members m and n to zero.

continue ...


Constructors

- ❖ There is no need to write any statement to invoke the constructor function.
- ❖ If a 'normal' member function is defined for zero initialization, we would need to invoke this function for each of the objects separately.
- ❖ A constructor that accepts no parameters is called the default constructor.

The default constructor for class ABA is:
ABC :: ABC ()

```
class add
{
    int m, n ;
public :
    void Initialize() //normal member function
    {
        m=0;
        n=0
    }
    .....
};

int main()
{
    add A,B,C;
    A.Initialize(); B.Initialize(); C.Initialize();
}
```



Characteristics of Constructors

- ❖ Constructors should be declared in the public section.
- ❖ They are invoked automatically when the objects are created.
- ❖ They do not have return types, not even void and they cannot return values.
- ❖ They cannot be inherited, though a derived class can call the base class constructor.
- ❖ Like other C++ functions, Constructors can have default arguments.
- ❖ Constructors can not be virtual.
- ❖ They make 'implicit calls' to the operators *new* and *delete* when memory allocation is required.
- ❖ When a constructor is declared for a class initialization of the class objects becomes mandatory.

Parameterized Constructors

- ❖ It may be necessary to initialize the various data elements of different objects with different values when they are created.
- ❖ This is achieved by passing arguments to the constructor function when the objects are created.
- ❖ The constructors that can take arguments are called parameterized constructors.

continue ...

Parameterized Constructors

```
class add
{
    int m, n ;
public :
    add (int, int) ;
    -----
};

add :: add (int x, int y)
{
    m = x; n = y;
}
```

- When a constructor is parameterized, we must pass the initial values as arguments to the constructor function when an object is declared.
- Two ways Calling: in main() function
 - Explicit
 - add obj = add(2,3); // like a function call
 - Implicit
 - add obj(2,3) // initialization while creating object
 - add obj1(a,b), obj2(a1,b1);
 - **add obj3; // will it allow me to create the object ?**

Multiple Constructors in a Class:

Also called as Constructor Overloading

- ❖ C++ permits to use more than one constructors in a single class to initialize the object with different values.
- ❖ `add() ; //` No arguments – It initialize object with default values (same values) or garbage values - **add obj3; - valid statement**
- ❖ `add (int, int) ; //` Two arguments – It initialize objects with parameterized values from calling function

continue ...

Multiple Constructors in a Class

```
class add
{
    int m, n ;
public :
    add ( ) {m = 0 ; n = 0 ;}
    add (int a, int b)
        {m = a ; n = b ;}
    add (add &obj)
        {m = obj.m ; n = obj.n ;}
};
```

- The first constructor receives no arguments.
- The second constructor receives two integer arguments.
- The third constructor receives one add object as an argument.

continue ...

Multiple Constructors in a Class

```
class add
{
    int m, n ;
public :
    add ( ) {m = 0 ; n = 0 ;}
    add (int a, int b)
    {m = a ; n = b ;}
    add (add &obj)
    {m = obj.m ; n = obj.n ;}
};
```

```
int main()
{
    add o1;
    add o2(10,20);
    add obj (o2);
    o1.print();
    o2.print();
    obj.print();
}
```

Add o1;

- Would automatically invoke the first constructor and set both m and n of a1 to zero.

Add o2(10,20);

- Would call the second constructor which will initialize the data members m and n of a2 to 10 and 20 respectively.

Add obj(o2);

Would call third constructor which will initialize data members of obj with values of o2 object.

This type of constructor is called the “copy constructor”.

Constructor Overloading

- More than one constructor function is defined in a class.

continue ...

Multiple Constructors in a Class

```
class complex
{
    float real, img;
public :
    complex ( ) {} //Default constructor
    complex (float a)
        { real = img = a ; }
    complex (float r, float i)
        { real = r ;img = i ;}
};
```

Do – nothing
Constructor

```
complex ( ) { }
```

- This contains the empty body and does not do anything.
- This is used to create objects without any initial values.
- Default constructor
- Example : `complex c1,c2,c3;`

continue ...

Multiple Constructors in a Class

```
class complex
{
    float real, img;
public :
    complex ( ) {} //Default constructor
    complex (float a)
        { real = img = a ; }
    complex (float r, float i)
        { real = r ;img = i ;}
};
```

- C++ compiler has an *implicit constructor* which creates objects, even though it was not defined in the class.
- This works well as long as we do not use any other constructor in the class.
- However, once we define a constructor, we must also define the “do-nothing” implicit constructor.

Constructors with Default Arguments

```
class complex
{
    float real, img;
public :
    complex ( ) {} //Default constructor
    complex (float r, float i=0.0)
        { real = r ;img = i ;}
};
```

Constructor with
default argument

- It is possible to define constructors with default arguments.
- Consider `complex (float real, float img = 0);`
 - The default value of the argument `img` is zero.
 - `complex C1 (5.0)` assigns the value 5.0 to the real variable and 0.0 to `img`.
 - `complex C2(2.0,3.0)` assigns the value 2.0 to `real` and 3.0 to `img`.

Example: read and print student data

```
class student
{
    char Name[40]; int RN;
    char Div;
public :
    student ( ) {} //Default constructor
    student (char n[], int r, char d = 'E')
    { strcpy (Name, n)
      RN = r;
      Div = d;
    }
    void display();
};
```

```
void student :: display ()
{
    cout << Name << RN << Div;
}

int main()
{
    student s("xyz",12);
    s.display();
}
```

Constructors with Default Arguments

- ❖ `A :: A ()` → Default constructor
- ❖ `A :: A (int = 0)` → Default argument constructor
- ❖ The default argument constructor can be called with either one argument or no arguments.
- ❖ When called with no arguments, it becomes a default constructor.

Dynamic Initialization of Objects

There are two kind of initializations:

Static initialization : Its either zero-initialization or initialization with a constant expression.

Any other initialization is **dynamic initialization**.

- ❖ Providing initial value to objects at run time.

Advantage –

- ❖ We can provide various initialization formats, using overloaded constructors.
- ❖ This provides the flexibility of using different format of data at run time depending upon the situation.

Using parameterized constructors

❖ The **Dynamic Initialization of Objects** means to initialize the data members of the class while creating the object.

❖ When we want to provide initial or default values to the data members while creating of object - we need to use **dynamic initialization of objects**.

```
class student
{
    char Name[40]; int RN;
    float per;
public:
    student()
    { }
    student(char n[], int rn, float p);
    void display();
};

student :: student(char n[], int rn, float p)
{
    strcpy (Name,n);
    RN = rn;
    per = p;
}
```

```
void student :: display()
{
    cout <<Name<<" "<<RN;
}

int main()
{
    char n[20]; int rn; float p;
    cin>>n>>rn>>p;
    student s(n,rn,p);
    s.display();
}
```

Using Copy Constructor

```
class add
{
    int m, n ;
public :
    add ( ) {m = 0 ; n = 0 ;}
    add (int a, int b)
    {m = a ; n = b ;}
    add (add &obj)
    {m = obj.m ; n = obj.n ;}
};
```

```
int main()
{
    int m1,n1;
    cin>>m1>>n1;
    add A(m1,n1);
    add A1(A);
    add A2 = A1;
}
```

❖ A copy constructor is used to declare and initialize an object from another object.

Example:

Complex (Complex & c);

Complex c2 (c1) ; or Complex c2 = c1 ;

- ❖ The process of initializing through a copy constructor is known as copy initialization.
- ❖ A reference variable has been used as an argument to the copy constructor.
- ❖ We cannot pass the argument by value to a copy constructor.

continue ...

Copy Constructor

❖ Some conceptual misunderstanding:

The statement

`Complex c1,c2; //declaring objects`

`c2 = c1; // will not invoke the copy constructor.`

❖ If `c1` and `c2` are objects, this statement is legal and assigns the values of `c1` to `c2`, member-by-member.

Dynamic Constructors

```
class student
{
    char *Name; int RN;

    public:

        student()
        {
        }

        student(char n[], int rn);
        void display();
};

void student::display()
{
    cout <<Name<<" "<<RN;
}
```

```
student::student(char n[], int rn)
{
    Name = new char[40];
    strcpy (Name,n);
    RN = rn;
}

int main()
{
    student s;
    student s1("xyz",10);
    //s.display();
    s = s1;
    s1.display();
    s.display();
}
```

- ❖ Allocation of memory to objects at the time of their construction is known as dynamic construction of objects.
- ❖ This will enable the system to allocate the right amount of memory for each object when the objects are not of the same size.
- ❖ When allocation of memory is done dynamically using dynamic memory allocator new in a constructor, it is known as **dynamic constructor**. By using this, we can dynamically initialize the objects.

Destructors

- ❖ A destructor is used to destroy the objects that have been created by a constructor.
- ❖ Like constructor, the destructor is a member function whose name is the same as the class name but is preceded by a tilde.

eg: `~ integer () { }`

continue ...

Destructors

- ❖ A destructor never takes any argument nor does it return any value.
- ❖ It will be invoked implicitly by the compiler upon exit from the program – or block or function as the case may be – to clean up storage that is no longer accessible.
- ❖ It is a good practice to declare destructors in a program since it releases memory space for further use.
- ❖ Whenever ***new*** is used to allocate memory in the constructor, we should use ***delete*** to free that memory.

Example 1:

```
#include<iostream>
using namespace std;
class Bank_Account
{
char cust_name[30];
long int account_No;
public:
Bank_Account()
{
cout<<"enter details of the customers : "<<endl;
cin>>cust_name;
cin>>account_No;
}
```

```
void print()
{
cout<<"details of the customers are:"<<endl;
cout<<cust_name<<endl;
cout<<account_No<<endl;
}
~Bank_Account()
{
cout<<"Destructor has been called"<<endl;
cout<<"Removed object (customer) from the
memory : "<<account_No<<endl;
}
};
```

```
int main()
{
Bank_Account C1;
C1.print();
{
Bank_Account C2;
C2.print();
}
{
Bank_Account C3;
C3.print();
}
Bank_Account C4,C5;
C4.print();
C5.print();
}
```

Example 1: Destructor

```
class student
{
    char *Name;

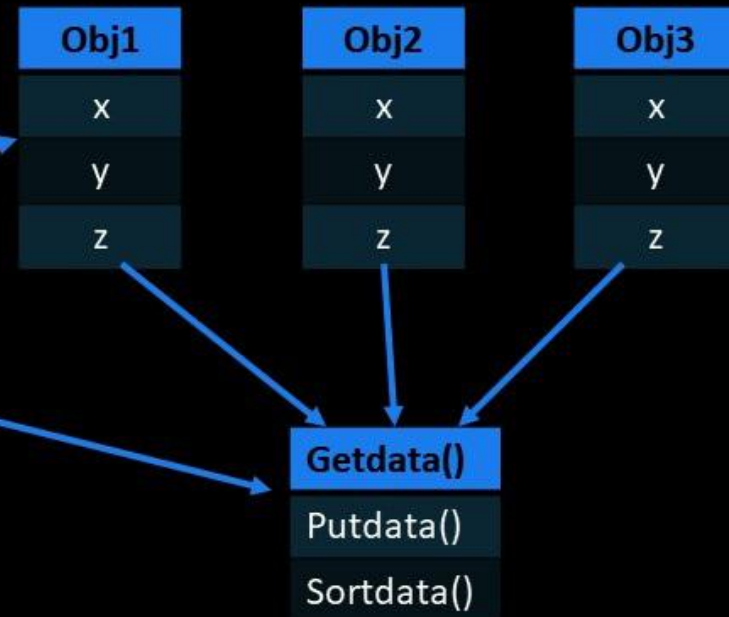
public:
    student( ) {} //Default constructor
    student(char n[])
    {
        Name = new char[20];
        strcpy(Name,n);
    }
    void display();
    ~student()
    {
        cout<<"destructor is called ";
        delete Name;
    }
};
```

```
void student :: display()
{
    cout <<Name<<" "<<endl;
}

int main()
{
    student s("xyz");
    s.display();
}
```

“this” pointer

- ❖ To understand ‘this’ pointer, it is important to know how objects look at functions and data members of a class.
- ❖ Each object gets its own copy of the data member.
- ❖ All-access the same function definition as present in the code segment.
- ❖ Meaning each object gets its own copy of data members and all objects share a single copy of member functions.



Then now question is that if only one copy of each member function exists and is used by multiple objects, how are the proper data members accessed and updated?

“this” pointer

Answer:

The compiler supplies an implicit pointer along with the names of the functions as ‘**this**’

In other words:

“**this**” pointer implicitly refers the data members and member functions of current object (i.e. the object which is calling a function)

Therefore, ‘this’ could be the reference than the pointer.

Syntax :

1. to access data members

this -> name;

2. to access member functions

this -> Getdata();

The **this** pointer is an implicit parameter to all member functions. Therefore, inside a member function, this may be used to refer to the invoking object.

Note:

Friend functions do not have a **this** pointer, because friends are not members of a class. Only member functions have a **this** pointer.

Example: "this" pointer

```
class Box
{
double length, breadth, height;

public: // Constructor definition
Box(double l = 2.0, double b = 2.0, double h = 2.0)
{
cout << "Constructor called." << endl;

length = l; breadth = b; height = h;
}

double Volume()
{
return length * breadth * height;
}

int compare(Box box)
{
return this->Volume() > box.Volume();
}
};
```

```
int main(void)
{
Box Box1(3.3, 1.2, 1.5);
Box Box2(8.5, 6.0, 2.0);

if(Box1.compare(Box2))
cout << "Box2 is smaller than Box1";
else
cout << "Box2 is equal to or larger than Box1";
}
```

this->Volume() : representing calling object function and operating of calling object (Box1) data members.

box.Volume() : representing explicit object (Box2) data members with same function Volume()

“this” pointer

If we do not use this pointer explicitly in the program then system uses it implicitly to represent current object.

However, it required to be use explicitly in following conditions:

1. When local variable's name is same as member's name

2. To return reference to the calling object

// Example: explicit use of this pointer **When local variable's name is same as member's name**

```
class ABC
{
    int x,y;
public:
    void getXY (int x, int y1)
    {
        // The 'this' pointer is used to retrieve the object's x
        // hidden by the local variable 'x'
        this->x = x;
        y = y1; // value of y1 is directly assigned without this pointer
                  since local variable name is different
    }
    void printXY() { cout << "x = " << x << endl; cout << "y = " << y << endl; }
};

int main()
{
    Test obj;
    int x = 20, y =30;
    obj.getXY(x,y);
    obj.printXY();
}
```

// Example: explicit use of this pointer **to return reference to the calling object for chain function call on single object**

```
class ABC
{
    int x,y;
public:
    void getXY (int x, int y)
    {
        this->x = x;          this->y=y;          }
    ABC &setX(int a) { x = a; return *this; }
    ABC &sety(int b) { y = b; return *this; }
    void printXY()
    {
        cout << "x = " << x << endl; cout << "y = " << y << endl; }
};

int main()
{
    ABC obj;
    obj.getXY(10,10);
    obj.printXY();

    obj.setX(20).sety(30); // chain function call using single object
    obj.printXY();
}
```

Output:

x = 10

y = 10

x = 20

y = 30