INDIAN INSTITUTE OF TECHNOLOGY ROORKEE

EEC-101
Programming with C++

Module-5:
Object Oriented Design:

About Subject

- Object Oriented Programming Concepts
  - Inheritance and composition;
  - Dynamic binding and virtual functions;
  - Polymorphism;
  - Dynamic data in classes.

Inheritance, when used to modify and extend the capabilities of the existing classes, becomes a very powerful tool for incremental program development.

Inheritance

- The reuse of a class that has already been tested, debugged, and used many times can save the effort of developing and testing the same again.
- C++ strongly supports the concept of ***reusability***

- Once a class is written and tested, it can be adapted by other programmers to suit their requirements.

- This can be done by creating new classes, reusing the properties of the existing ones.

- The mechanism of deriving a new class from an old one is called **inheritance (or derivation).**

Base and Derived Class

- The old class is referred to as the **base class.**
- The new one is called the **derived class** or **subclass.**
- The derived class inherits some /all the traits from the base class.
- A class can also inherit properties from more than one class or from more than one level.

Types of Inheritance

1. A derived class with only one base class, is called **single inheritance**.

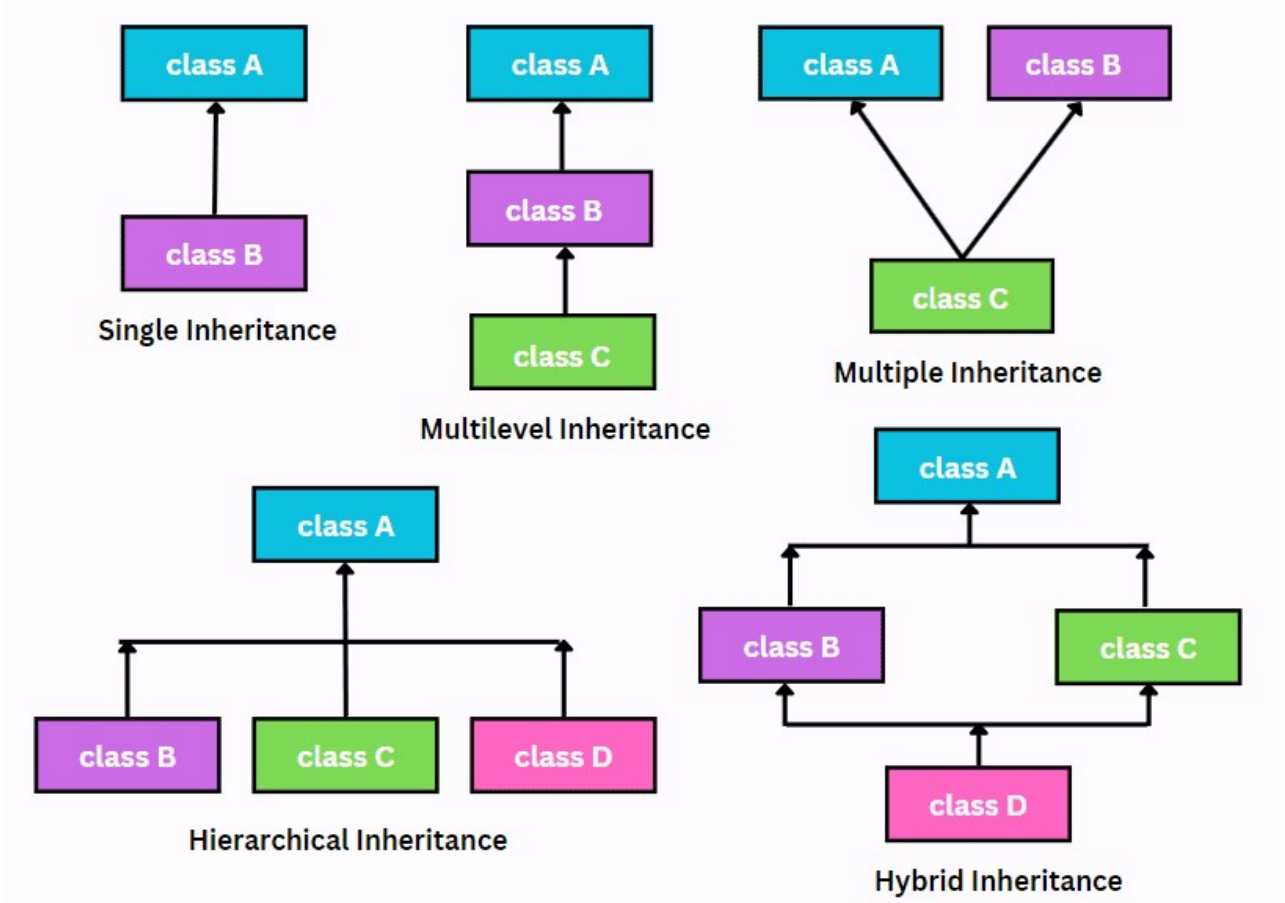
2. The mechanism of deriving a class from another 'derived class' is called **multilevel inheritance**.

3. Traits of one class may be inherited by more than one class. This process is referred to as **hierarchical inheritance.**

4. A derived class with several base classes is called **multiple Inheritance.**

5. Where two or more types of inheritance are applied to design a program is called **hybrid inheritance.**

Types of Inheritance



Defining Derived Classes

- A derived class can be defined by specifying its relationship with the base class in addition to its own details.

Syntax:

class *derived-class-name* : *visibility-mode* **base-class-name**

{

……

…… // members of the derived class

……

}

- *colon* indicates that the derived-class-name is derived from the base class
- *visibility-mode* is optional, by default it is private.
- if present, it may be public, protected, or private

Example:

```
class ABC: private XYZ  //private  derivation
{
     members of ABC
};

class ABC: public XYZ    //public derivation by
 {
   members of ABC
};

class ABC: XYZ  //private  derivation  by default
{
     members of ABC
};
```

(in all the cases, XYZ is the base class and ABC is the derived class)

- When a base class is **privately inherited by a derived class:**

  – public members of the base class become private members of the derived class

- When the base class is **publicly inherited:**
  - public members of the base class become public members of the derived class
  - Therefore, they are accessible to the objects of the derived class
- **Private members are not inherited**
- And therefore, **private members of a base class will never become members of its derived class**

Single Inheritance

Example

```
//Single Inheritance with public derivation
#include <iostream>
using namespace std;
class B
{
private:
int a;
// private: not inheritable

public:
// public: ready for inheritance
int b;
void get_ab();
int get_a();
void show_a();
};
class D: public B
// public derivation
{
private:
    int c;
public:
    void mul (void);
    void display (void);
};

void B::get_ab (void)
            {a=5; b=10; }
int B::get_a()
            { return a; }
void B::show_a()
          { cout<<"a= "<<a<<"\n";}
void D:: mul ()
          {c=b*get_a(); }
void D:: display ()
{
cout<<"\n"<<"a= "<<get_a() <<"\n";
cout<<"b= "<<b<<"\n";
cout<<"c= "<<c<<"\n";}

int main()
{
D d; // d object of derived class D
d.get_ab();// d is accessing public member of base class B
d. mul();// d is accessing public member of its own class D
d.show_a();// d is accessing public member of base class B
d.display(); // d is accessing public member of its own class D
d. b=20;// b is public member of class B, hence accessible outside class
d. mul();
d. display();
system ("pause");
return 0;} //end main
```

```
a= 5

a= 5
b= 10
c= 50

a= 5
b= 20
c= 100
Press any key to continue . . .
```

- Private member of a base class cannot be inherited, and therefore it is not available for the derived class directly.
- This requirement can be accomplished by modifying the visibility limit of the **private member** by making it **public.**

- However, this would make it accessible to all other functions of the program, **taking away the advantage of data hiding**.

- C++ provides a third *visibility modifier*, **protected**

Protected

- A member declared as **protected** is accessible by the member functions within its class and any class **immediately derived from it.**

- It cannot be accessed by the functions outside these two classes.

- A class can use all the three visibility modes as shown in the example:

```
class  A
{
private:
    …….    // optional
    …….   //  visible to member functions within its class
protected:
    ……..
    …….. //visible to member functions of its own and derived class
public:
    ……..
    ……..  // visible to all functions in the program
};
```

- When a **protected member** is **inherited in public mode**, **it becomes protected in the derived class**, too, and, therefore, is accessible by the member functions of the derived class.

- It is also ready for further inheritance.

- **Protected member inherited in private mode** derivation, becomes **private in the derived class**

- Although it is available to the member functions of the derived class, **it is not available for further inheritance.**

- Since private members cannot be inherited.

- It is also possible to inherit a base class in **protected mode**, called **protected derivation**

- In **protected derivation**, **both public and protected members of the base class** become **protected members** of the **derived class**

Visibility of Inherited members

Access mode in Base class	Class Inherited as:	Resulting Access mode in Derived class
Public	Public	Public
Protected		Protected
Private		Not Inherited
Public	Protected	Protected
Protected		Protected
Private		Not Inherited
Public	Private	Private
Protected		Private
Private		Not Inherited

Example

```
/*Program to check the accessibility of data members
and member functions of base class from derived class
 member functions or from objects of derived class */
#include<iostream>
using namespace std;
class one //base class
{
private:
    int a=1;
protected:
    int b=2;
public:
    int c=3;
    };
class two: public one // publicly derived class
{
public:
void function1()
{
int z=0;
//z=a; // error: not accessible;
z = b; // works
z = c; // works
}
};
```

Base Class Visibility	Derived Class visibility	
	Public derivation	Private derivation
Private	Not inherited	Not inherited
Protected	Protected	Private
Public	Public	Private

```
class three: private one // privately-derived class
{
    public:
    void function2 ()
    {
    int y=5;
//    y= a; // error: not accessible
    y = b; // works
    y =c; // works
    }
};
int main()
{
int x=0;
two second;// object of class two
//x=second.a;// error: not accessible
//x=second.b;// error: not accessible
    x= second.c;// works
    cout<<x<<endl;
three third;// object of class three
//x=third.a;// error: not accessible
//x=third.b;// error: not accessible
//x=third.c/ error: not accessible //
system ("pause");
return 0;}
```

Base Class Visibility	Derived Class visibility	
	Public derivation	Private derivation
Private	Not inherited	Not inherited
Protected	Protected	Private
Public	Public	Private

Reasoning for the Previous Example

```
void function1()
{
int z;
//z = a; // error: not accessible; a is private in base class so not inherited in derived class
z = b; // works: b is protected in publicly derived class (accessible in its own class and derived)
z = c; // works: c is public in base class also in publicly derived class, c is remains public
}
```

```
class three: private one // privately-derived class
{
public:
void function2()
{
int y;
//y = a;    // error: not accessible; a is private in base class so not inherited
y = b;  // works: b is protected in base class, now in privately derived class becomes private
        // though b is now private in derived class, accessible within class
y = c; // works: c is public in base class, now in privately derived class becomes private
        //though c is now private in derived class, accessible within class
}
```

Reasoning for the Previous Example

```
int main()
{
int x;
two second;          // object of class two
//x = second.a; // error:not accessible, since a is private and not inherited
//x= second.b;  // error: not accessible, b is protected (accessible in base class and derived class)
                // b in protected mode in base class derived publicly
                //is not accessible outside base & derived class
x = second.c; // works: C is public, hence inherited and accessible outside base and derived class

three third;// object of class three (privately derived)
//x = third.a;  // error:not accessible, a is private in base class so not inherited
//x = third.b;  // error: not accessible, b is protected in base class,
                //b becomes private in derived class three, so not accessible outside class
//x= third.c // error: not accessible, c is public in base class,
            // c becomes private in derived class three as it is derived privately
system("pause"); return 0;}
```

Example

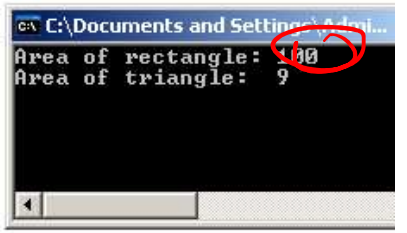
```
//Inheritance example - public derivation
#include<iostream>
using namespace std;
class Cpolygon // Base class
{
    protected:
        int width, height;
    public:
        void input_values (int one, int two)
        { width=one; height=two;}
};
class Crectangle: public Cpolygon // public derivation
{
    public:
        int area ()
        { return (width * height);}
};
class Ctriangle: public Cpolygon // public derivation
{
    public:
        int area ()
        { return (width * height / 2);}
};
```

```
int main ()
{
    Crectangle rectangle;
    Ctriangle  triangle;

    rectangle.input_values(5,20);
    triangle.input_values(3,6);

    cout <<"Area of rectangle: "<< rectangle.area() << endl;
    cout <<"Area of triangle:  "<< triangle.area() << endl;

    cin.get();
    return 0;}
```



Friend Class

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

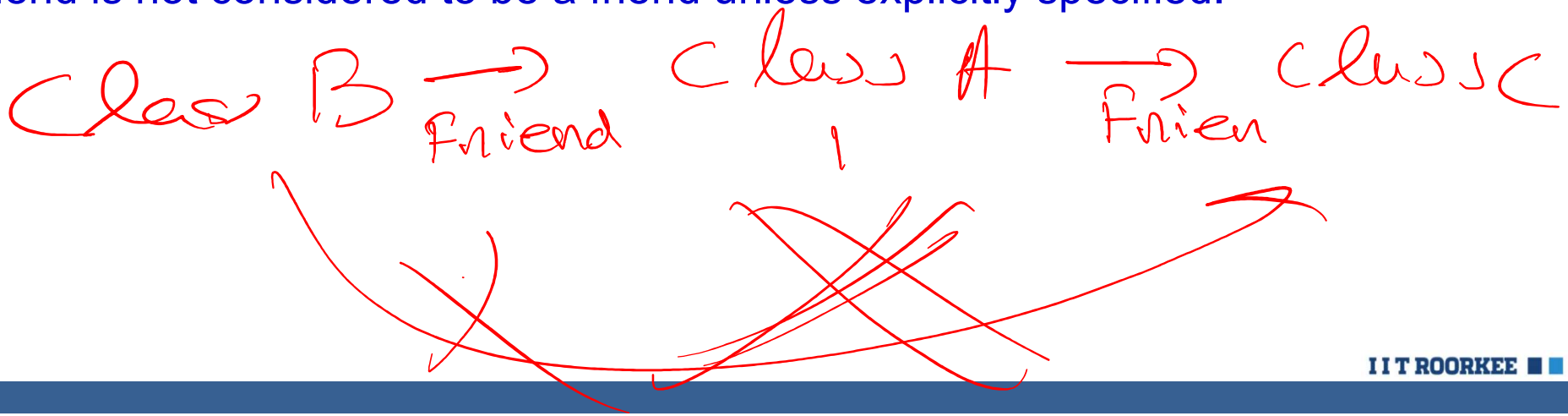
- We can declare a friend class in C++ by using the friend keyword.

- Syntax:

```
friend class class_name;    // declared in the base class
```

Friend Class

- When we create a friend class then all the member functions of the friend class also become the friend of the other class.
- All member functions of a friend class can access the private data of the class to which it is a friend.
- Another property of friendships is that they are not transitive, i.e., The friend of a friend is not considered to be a friend unless explicitly specified.



Example: Friend function to two classes

```
//Friend function to two classes
#include<iostream>
using namespace std;

class One; // forward declaration

class Two {
    int y;

    public:
    void setvalue (int i){y = i;}
    friend void max (One, Two);
// Declaration of friend function with arg. from both the classes
};
class One {
    int x;

    public:
    void setvalue (int i) {x = i;}
friend void max (One, Two);
//Declaration of friend function
};
```

```
// Definition of friend function
void max (One obj1, Two obj2){
if (obj1.x >= obj2.y)
cout<<"Maximum value is: "<< obj1.x << endl;
else
cout<<"Maximum value is: "<< obj2.y<< endl;
}
int main(){
One n1; // Create an object of type One
n1.setvalue (77);
Two t1; // Create an object of type Two
t1. setvalue (99);
max (n1,t1);
// call the friend function (not invoked by any object)
system ("pause");
return 0;}
```

```
Maximum value is: 99
Press any key to continue . . .
```

```
//Friend Classes (Class friendship is not reciprocal)
#include<iostream>
using namespace std;
//class One granting friendship to class Two
// members functions of class two can access members of class one
//member functions of class One cannot access the members of friend class Two

class One {
private:
int x; // private data
friend class two; // Declare a friend class
public:
One () { x = 99;} // Default Constructor
};

//class Two granted friendship
//can have unrestricted access to the members of the class One (granting the friendship)
class two{
public:
int subtract_from (int n)
// member function accessing private members of class one
{ One obj1;
return obj1.x - n;}
};

int main()
{
two obj2; // Create an object of class Two
cout << "Subtracted Result is : "<< obj2.subtract_from (33) <<endl;
system ("pause"); return 0; }
```

Two is friend with one

→ Two can access private members of one

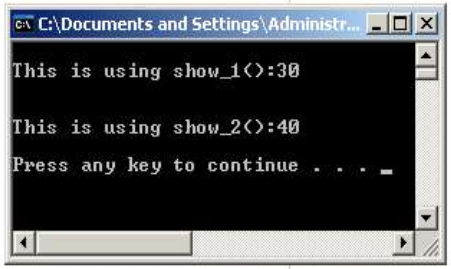
```
Subtracted Result is : 66
Press any key to continue . . .
```

```
//Example of Friend Class for unrelated classes
#include <iostream>
using namespace std;
class One // Class granting freindship
{
    private:
        int x;
        int y;
    public:
        One(int xx=0,int yy=0){ x = xx; y = yy;}    // Constructor with default argument
        friend class Two;// Declare a friend class
};
class Two// Class granted friendship
{
public:
void show_1(One obj1)
{ cout<<"\nThis is using show_1():"
<<obj1.x<<endl<<endl;}// x is private member of class One

void show_2(One obj1)
{ cout <<"\nThis is using show_2():"
<<obj1.y<<endl<<endl;}// y is private member of class one
};
int main()
{
One a(30,40);// Declared and initialzed using class one constructor
Two b;

b.show_1(a);//All member functions of freind class can access private data of One
b.show_2(a);//All member functions of freind class can access private data of One

system("pause"); return 0;}
```



```
// friend class Example
#include <iostream>
using namespace std;
class CSquare; //Forward Declaration

class CRectangle {
    int width, height;
    public:
    int area () {return (width *height); }

// prototype of member function accessing members of friendship granting class CSquare
    void convert (CSquare a);
};

class CSquare {
    private:
    int side;
    public:
    void set_side (int a) {side=a;}
    friend class CRectangle; // class CRectangle granted friendship
};

 //Definition of member function of class CRectangle granted friendship accessing
 //private members of class CSquare granting friendship to class CRectangle
void CRectangle:: convert (CSquare a)
{width = a.side + 2; height = a.side; }

int main() {
CSquare sqr;
CRectangle rect;
sqr.set_side (4);
rect.convert (sqr);
cout << rect. area ();
cout<<endl;
system ("pause"); return 0;}
```

Access control to private and protected

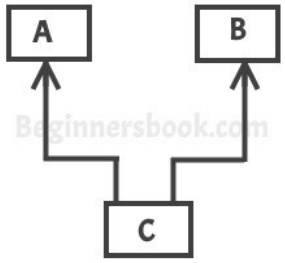
- Functions that can have access to private/protected members are :
  1. A function that is a friend of the class
  2. A member function of a class that is a friend of the class
  3. A member function of a derived class

- The friend function and member function of a friend class can have direct access to private and protected data

- Member functions of the derived class can directly access only the protected and public data. They can access the private data through the member functions (public) of the base class

Multiple Inheritance

- A class can inherit the attributes of two or more classes
- This is known as **multiple inheritance**
- Syntax of a derived class with multiple base classes :



Multiple Inheritance

class D: visibility BASECLASS1, visibility BASECLASS2, ...
{
.....
..... (Body of D)
.....
};

```
//Multiple Inheritance
#include <iostream>
using namespace std;
class M  // First Base class
{ protected:int m;
  public:void get_m(int);
};
class N  // Second Base Class
{ protected:int n;
  public:void get_n(int);
};
// Class P would contain all the members of class M & N
class P: public M, public N
// Derived from multiple classes i.e M & N
{ public:void display(void);
};
void M::get_m(int x)
{ m=x;}
void N :: get_n(int y)
{ n=y;}
void P :: display(void)
// mem. fun. of derived class P accessing protected members of M & N
{    cout<<"m= "<<m<<"\n";
     cout<<"n= "<<n<<"\n";
     cout<<"m*n= "<<m*n<<"\n";
}
int main()
{
     P p; // Declaration of derived class object
     p.get_m(10);// derived class object invoking mem. func of first base class M
     p.get_n(20);// derived class object invoking mem. func of second base class N
     p.display();//derived class object invoking mem. func of its own base class P
  system("pause");return 0;}
```

```
m= 10
n= 20
m*n= 200
Press any key to continue . . .
```

```
// Program to illustrate multiple inheritance
#include <iostream>
using namespace std;
class CPolygon // First Base Class
{
    protected:
    int width, height;
    public:
    void set_values (int a, int b) { width=a; height=b; }
};
class COutput // Second Base Class
{ public:
void output (int i);
};
void COutput::output (int i) {cout << i << endl; }

// Derived Class CRectangle would contain all the members of CPolygon and Coutput
class CRectangle: public CPolygon, public COutput{
    public:
    int area () { return (width* height); }
};

// Derived Class CTriangle would contain all the members of CPolygon and Coutput
class CTriangle: public CPolygon, public COutput // Derived Class
{
    public:
    int area () {return (width *height / 2); }
};
int main(){
CRectangle rect; // Object of derived class CRectangle instantiated
CTriangle trgl; // Object of derived class CTriangle instantiated
rect.set_values (5,10); //object of derived class invoking member function of first base class
trgl.set_values (5,10); //object of derived class invoking member function of first base class
rect.output (rect. area()); //object of derived class invoking its own mem. func and then of second base class
trgl.output (trgl. area ()); //object of derived class invoking its own mem. func and then of second base class
system ("pause");
return 0;}
```

```
50
25
Press any key to continue . . .
```

Function Overriding

- Derived class can override a base-class member function with the same signature
- It provides a new version of the function

- It is different from function overloading
  - In overloading, two functions have the same name but different parameter list
  - **In overriding, the functions have the same name and the same parameter list**

- For a derived class object, call to such a function invokes the derived class version
- Original function can be invoked through scope resolution operator

```
//Functions with same name in more than one class
#include<iostream>
using namespace std;
class M //Base Class
{
    public:
    void display (void) { cout<<"Class M\n"; }
};
class N // Base Class
{
    public:
    void display (void) { cout<<"Class N\n"; }
};
class P: public M, public N // Derived Class from base classes M and N
{
    public:
    void display (void){cout<<"Class P\n";}
};
int main()
{
P p;
p.display(); // Overrides the display () of M and N Classes
system ("pause");
return 0;
}
```

```
Class P
Press any key to continue . .
```

```
////Example 2
//Function Overriding in multiple inheritance
//Functions with same name in more than one class
#include<iostream>
using namespace std;
class M //Base Class
{ public:
void display (void) { cout<<"Class M\n";}
};
class N // Base
{ public:
void display (void) { cout<<"Class N\n"; }
};
class P: public M, public N // Derived Class from base classes M and N
{ public:
void display (void){
cout<<"Class P\n";
M::display (); // accessing display() of class M
N::display(); // accessing display() of class N
}};
int main(){
P p;
p.display(); // Overrides the display () of M and N Classes
system ("pause");return 0;}
```

```
Class P
Class M
Class N
Press any key to continue . . .
```

```cpp
//Example 3 Function Overriding in multiple inheritance
//Functions with same name in more than one class
//using scope resolution operator
#include<iostream>
using namespace std;
class M //Base Class
{
public:
void display (void) { cout<<"Class M\n"; }
};
class N // Base
{ public:
void display (void) { cout<<"Class N\n"; }
};
class P: public M, public N
// Derived Class from base classes M and N
{ public:
void display (void)
{cout<<"Class P\n";}
};

int main()
{
P p; // Derived Class object
p.display(); // Overrides the display () of M and N Classes
p. M::display (); // invokes display () of Class M(using scope resolution operator)
p.N::display (); // invokes display () of Class N (using scope resolution operator)
p. P::display (); // invokes display() of Class P(using scope resolution operator)
system ("pause");
return 0;
}
```

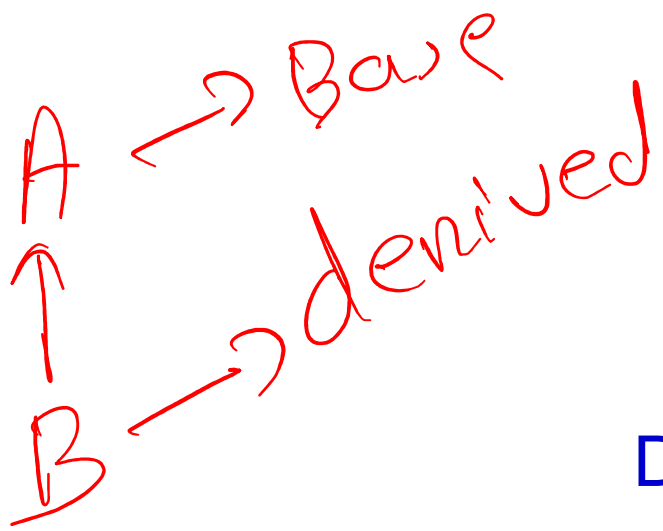
```
Class P
Class M
Class N
Class P
Press any key to continue . . .
```

Multilevel Inheritance

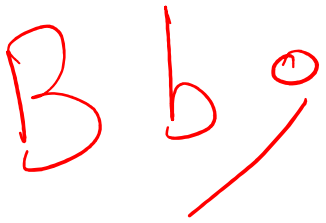
- Class **A** serves as base class for derived class **B**
- Class **B** in turn serves as base class for class **C**
- Class **B** is called **intermediate base class** because it provides a link for the inheritance between **A** and **C**
- The chain **ABC** is called as the **inheritance path**

- A derived class with **Multilevel Inheritance** is declared as follows:

```
class A{…….};  // Base class
class B: public A {……};  // B derived from A
class C: public B {…….}; //C derived from B
```



Constructors in Derived Classes



What is not inherited from the base class?

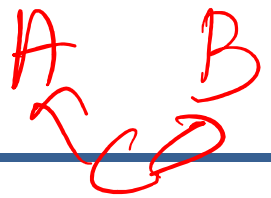
- Constructors
- Destructors
- Non-member functions
- Assignment operators
- Virtual methods

If needed, they must be created

- Whenever we create an object of a class, the default constructor of that class is invoked automatically to initialize the members of the class.

- If we inherit a class from another class and create an object of the derived class, it is clear that the default constructor of the derived class will be invoked but before that, the default constructor of all of the base classes will be invoked,

- i.e., the order of invocation is that the base class's default constructor will be invoked first, and then the derived class's default constructor will be invoked.



Why the base class's constructor is called to create an object of the derived class?

- To understand this you will have to recall your knowledge on inheritance. What happens when a class is inherited from another?
- The data members and member functions of the base class come automatically in the derived class based on the access specifier, but the definition of these members exists in the base class only.
- So when we create an object of the derived class, all of the members of the derived class must be initialized, but the inherited members in the derived class can only be initialized by the base class's constructor, as the definition of these members exists in the base class only. This is why the constructor of the base class is called first to initialize all the inherited members.

- Although the constructors and destructors of the base class are not inherited,
  - its default constructor (i.e., its constructor with no parameters) and
  - its destructor are always called when a new object of a derived class is created or destroyed.

- In the case of multiple inheritance, base classes are constructed in the **order in which they appear** in the declaration of the derived class.

- In the case of **multilevel inheritance, the constructor will be executed in the order of inheritance**.

```cpp
// C++ program to show the order of
//constructor call in single inheritance
#include <iostream>
using namespace std;
// base class
class Parent
{    public:
        // base class constructor
    Parent(){
    cout << "Inside base class" << endl; }
};
// sub class
class Child : public Parent
{    public:
        //sub class constructor
    Child(){
    cout << "Inside sub class" << endl;}
};
// main function
int main() {
        // creating object of sub class
    Child obj;
    return 0; }
```

```
Inside base class
Inside sub class

Process returned 0 (0x0)   executior
Press any key to continue.
```

Constructor call for Multiple Inheritance

For multiple inheritance, the base class's constructors are called in the order of inheritance, and then the derived class's constructor.

class A : public C, public B

Order of Inheritance

Class C (Base Class 2)
↓
Class B (Base Class 1)
↓
Class A (Derived Class)

Order of Constructor Call

1. C() (Class C's Constructor)
2. B() (Class B's Constructor)
3. A() (Class A's Constructor)

Order of Destructor Call

1. ~A() (Class A's Destructor)
2. ~B() (Class B's Destructor)
3. ~C() (Class C's Destructor)

How to call the parameterized constructor of base class in derived class constructor?

- To call the parameterized constructor of base class when derived class's parameterized constructor is called, you have to explicitly specify the base class's parameterized constructor in derived class

- The header of the derived-constructor function contains two parts separated by a colon(:).

  - The first part provides the declaration of arguments that are passed to the derived constructor
  - The second part lists the function calls to the base constructors

```
derived_constructor_name (parameters) : base_constructor_name (parameters)
{
 Body of derived constructor
}
```

Example:

Constructor for derived class D derived from Class A and Class B

```
D (int a1, int a2, float b1, float b2, int d1): A(a1, a2), B(b1,b2)
{
d = d1;
}
```

A(a1,a2) invokes base constructor A()
B(b1,b2) invokes another base constructor B()

Constructor D supplies the values for these four arguments and the fifth value is passed to its body

```
D obj(5,10,2.5,8.65,30);
```

Values assigned:

a1 -> 5 a2 -> 10 b1->2.5 b2->8.65 d->30

- Derived class takes the responsibility of supplying initial values to its base classes
- We supply the initial values that are required by all the classes together when a derived class object is declared.
- The constructor of the derived class receives the entire list of values as its arguments and passes them on to the base constructors **in the order in which they are declared in the derived class.**
- The base constructors are called and executed before executing the statements in the body of the derived constructors.

```
// Example 1 - Constructors and derived classes
#include <iostream>
using namespace std;
class Base {
    public:
    Base (){        // default constructor
    cout << "Base Class default: no parameters\n"; }
    Base (int a) { //parameterized constructor
    cout << "Base Class: int parameter\n"; }
};
class Done :public Base {
public:
Done (int a)
{ cout << "Derived Class DOne: int parameter\n\n"; }
// constructor in derived class
};

class DTwo: public Base {
public:
DTwo (int a):Base (a)
// constructor in derived class
{ cout << "Derived Class DTwo: int parameter\n\n"; }
};
```

```
int main() {
Done d1(0);
DTwo d2 (0);
system ("pause");
return 0;
}
```

```
Base Class default: no parameters
Derived Class DOne: int parameter

Base Class: int parameter
Derived Class DTwo: int parameter

Press any key to continue . . .
```

Object Compositions

- An object is a basic unit of Object-Oriented Programming and represents real-life entities. Complex objects are objects that are built from smaller or a collection of objects.
- For example, a mobile phone is made up of various objects like a camera, battery, screen, sensors, etc. This process of building complex objects from simpler ones is called object composition.

- In object-oriented programming languages, object composition is used for objects that have a "has-a" relationship with each other.
- Therefore, the complex object is called the whole or a parent object whereas a simpler object is often referred to as a child object.

Composition

is-a relationship

- describes a has-a relationship
- a class contains objects of some other classes as members
- e.g. an employee class may have Date of birth and phone Number as members

Example

```
class employee {
   private:
        char name[30];
        date Dob;
        phone officePhone;
   ....
};
```

Dob is an object of class date

Composition

Constructor for employee class

```
employee::employee (char * s, int bd, int bm, int by, int std, int number) : Dob(bd, bm, by),
   officePhone(std, number)
{
   //code for setting the name part
}
```

- if a member initializer is not provided, then member object's default constructor is called implicitly.

```
#include <iostream>
using namespace std;
class A {  // Simple class
public:
    int x;
    // Constructor initializing the data members
    A() { x = 0; }
    A(int a){
        cout << "Constructor A(int a) is invoked" << endl;
        x = a;}
};
class B { // Complex class
    int data;
    A objA;
public:
    // Constructor initializing the data members
    B(int a): objA(a){
        data = a;
    }
// Function to print values of data members in class A and B
    void display() {
    cout << "Data in object of class B = " << data<< endl;
    cout << "Data in member object of "
            << "class A in class B = " << objA.x;
    } };
// Driver code
int main(){
    B objb(25);// Creating object of class B
    objb.display();// Invoking display function
    return 0;
}
```

```
Constructor A(int a) is invoked
Data in object of class B = 25
Data in member object of class A in class B = 25
Process returned 0 (0x0)   execution time : 0.088 s
Press any key to continue.
```

Pointer and Inheritance

- A derived class object is a base class object also.
- Hence, a pointer to a base class object can be made to point to a derived class object.
- Thus, derived class and base class objects can be handled through a common pointer.

Thanks