AMBITION GURU

C++ Programming (HCAC-154)

II semester, BCA

Compiled by Ankit Bhattarai
Ambition Guru College

Syllabus

AMBITION GURU

Unit	Contents	Hours	Remarks
1.	OOP Basics	9	
2.	Classes & Objects	10	
3.	**Overloading and Inheritance**	**10**	
4.	Pointers	8	
5.	Virtual function and polymorphism	8	
6.	Templates and Exception Handling	7	
7.	I/O stream	8	

Practical Works

Credit hours : 3

AMBITION GURU

Unit 3

Overloading and Inheritance

(10 hrs.)

Operator Overloading: Syntax of Operator Overloading, Overloading unary operator: Operator Keyword, Operator arguments, Operator return value, , overloading binary operator, arithmetic operators, comparison operator, arithmetic assignment operator, data conversion; conversion between objects of different classes, Operator Overloading with Member and Non-Member Functions

Inheritance: Derived Class & Base Class: Specifying the Derived class accessing Base class members, the protected access specifier, Derived class constructor, Overriding member functions, public and private inheritance; Access Combinations, Classes & Structures, Access Specifiers. Multilevel inheritance, Hybrid inheritance, Multiple inheritance; member functions in multiple inheritance, constructors in multiple inheritance, Containership; Classes, within classes, Inheritance & Program development.

AMBITION GURU

Operator Overloading: Syntax of Operator Overloading, Overloading unary operator: Operator Keyword, Operator arguments, Operator return value, , overloading binary operator, arithmetic operators, comparison operator, arithmetic assignment operator, data conversion; conversion between objects of different classes, Operator Overloading with Member and Non-Member Functions.

Operator Overloading :

AMBITION GURU

- The process of redefining or extending the meaning of the existing operators while using with user defined data type is called *operator overloading*.
- The operator overloading feature of C++ is one of the methods of realizing polymorphism.

Following are the operators that cannot be overloaded in C++:

1. Member access operators (.)
2. Scope resolution operator(::)
3. Conditional operator(?:)
4. Size operator (sizeof( ))
5. Run-time type information operator (typeid)
6. Pointer to member access (*)

AMBITION GURU

Operator Overloading:

Consider the following statements

```
int a,b;

int  c = a + b;

complexx c1,c2;

complexx c = c1 + c2;
```

- Here **+ operator** is said to be overloaded as it is used with basic type data a and b as well as user defined type c1 and c2.
- Other examples can be c1+5,c1++,++c1 etc. where c1 is the object of type complexx.

Rules for operator overloading:

i. Only the existing operators can be overloaded, new operators cannot be created.

ii. The overloaded operator must have at least one operand that is of user defined type.

iii. We cannot change the basic meaning of the operators that is to say we cannot redefine (+) operator to subtract one value from the another.

iv. Unary operator overloading through member operator function takes no argument where as overloading through non-member operator function(friend function) takes one argument.

v. Binary operator overloading through member operator function takes one argument where as overloading through non-member operator function takes two arguments.

vi. We cannot use friend function to overload the following operators:

1. = (assignment operator)
2. ( ) (function call operator)
3. [ ] (subscripting operator or index operator)
4. -> (arrow operator)

General syntax of operator overloading

```
class class_name{
   // private data members;
   public:
   return_type operator operator_symbol(no argument or argument);
   Or, // member operator function
   friend return_type operator operator_symbol(argument(s)); // friend function
};
```

```
 return_type class_name::operator operator_symbol( no argument or argument)
{
     //Body of member operator function.
}

Or,

return_type operator operator_symbol(argument(s))
{
    //Body of non-member operator function(friend function).
}
```

Here,

return_type is the return type of the function.

operator is a keyword.

operator_symbol is the operator we want to overload. Like: +, <, -, ++, etc.

arguments is the arguments passed to the function.

AMBITION GURU

Example 1:
Add Two Complex Number

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

class Complex {
private:
    float real;
    float imag;

public:
    Complex(float r = 0, float i = 0) {
        real = r;
        imag = i;
    }

 // Operator overloading of +
    Complex operator + (Complex c) {
        Complex temp;
        temp.real = real + c.real;
        temp.imag = imag + c.imag;
        return temp;
    }

    void display() {
        cout << real << " + " << imag << "i" << endl;
    }
};
```

AMBITION GURU

Example 1:
Add Two Complex Number

```
int main() {
    Complex c1(2.5, 3.5), c2(1.5, 2.5), c3;

    c3 = c1 + c2;  // Using overloaded +

    cout << "Sum: ";
    c3.display();

    return 0;
}
```

AMBITION GURU

The operator Keyword

How do we teach a normal C++ operator to act on a user-defined operand?

- The keyword operator is used to overload the ++ operator in this declarator:

```
void operator ++ ()
```

- The return type (void in this case) comes first, followed by the keyword operator, followed by the operator itself (++), and finally the argument list enclosed in parentheses (which are empty here).
- This declarator syntax tells the compiler to call this member function whenever the ++ operator is encountered, provided the operand (the variable operated on by the ++).

Operator Arguments

- In main() the ++ operator is applied to a specific object, as in the expression ++c1.

Yet operator++() takes no arguments. What does this operator increment?

- It increments the data in the object of which it is a member.
- Since member functions can always access the particular object for which they’ve been invoked, this operator requires no arguments.

AMBITION GURU

Friend function in C++

Friend function in C++

- In C++, a ***friend function*** is a function that is **not a member of a class**, but still has access to the class's **private and protected** members.
- It is useful when you want a function (global or member of another class) to access internal members of a class without making them public.
- A friend function is declared inside the class using the keyword friend.
- It is not a member function of the class.
- It can access private and protected members of the class.
- Can be a normal function, or a member of another class.

Friend function in C++

Why use it?

- To allow an external function to **access private data** of a class.
- Useful when you want a function to **operate on multiple objects** (possibly from different classes).

Friend function in C++

AMBITION GURU

Syntax of Friend Function

Inside class:

```
friend returnType functionName(arguments);
```

Outside class:

```
returnType functionName(arguments) {
    // can access private members
}
```

Example 1: Add Two Numbers Using Friend Function

AMBITION GURU

Friend function in C++

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

class Number {
private:
    int value;

public:
    Number(int v = 0) {
        value = v;
    }

    // Declare friend function
    friend int add(Number a, Number b);
};

// Define friend function
int add(Number a, Number b) {
    return a.value + b.value;
}

int main() {
    Number n1(10), n2(20);

    cout << "Sum = " << add(n1, n2) << endl;

    return 0;
}
```

Friend function in C++

AMBITION GURU

Example 2: Friend Function Accessing Two Different Classes

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

class B;   // Forward declaration

class A {
private:
    int a;

public:
    A(int val = 0)  {
        a = val;
    }

    // Declare friend function
    friend int add(A, B);
};
```

Friend function in C++

AMBITION GURU

Example 2: Friend Function Accessing Two Different Classes

```
class B {
private:
    int b;

public:
    B(int val = 0) {
        b = val;
    }

    // Declare friend function
    friend int add(A, B);
};

// Define the friend function
int add(A objA, B objB) {
    return objA.a + objB.b;
}
```

Friend function in C++

AMBITION GURU

Example 2: Friend Function Accessing Two Different Classes

```
int main() {
    A obj1(10);
    B obj2(20);

    cout << "Sum = " << add(obj1, obj2) << endl;

    return 0;
}
```

Output:

```
Sum = 30
```

Unary operator overloading:

- If the existing operator is used within the single operand it is called unary operator.
- Unary prefix and postfix operators can be overloaded with the help of member operator function as well as friend function.

Syntax for overloading unary prefix and unary postfix operators through member operator function:

```
class class_name{
  //private data members;
  public:
  return_type operator operator_symbol ( );  //for prefix
  return_type operator operator_symbol(int);  //for postfix
};
```

```
return_type class_name::operator operator_symbol() //prefix
{
    //Body of member operator function.
}

return_type class_name::operator operator_symbol(int) //postfix
{
    //Body of member operator function.
}
```

AMBITION GURU

Syntax for unary prefix and postfix overloading with the help of friend function:

```
class class_name
{
//private data_members;
public:
friend return_type operator operator_symbol(class_name); //prefix
friend return_type operator operator_symbol(class_name,int); //postfix
};
```

```
return_type operator operator_symbol (class_name object_name)//prefix
{
//Body of friend function.
}


return_type operator operator_symbol(class_name object_name,int)//postfix
{
//Body of friend function
}
```

```
// Overload ++ when used as prefix
#include <iostream>
using namespace std;
class Count {
   private:
   int value;
   public:
    // Constructor to initialize count to 5
    Count(){
        value=5;
    }
    // Overload ++ when used as prefix
    void operator ++ () {
        ++value;
    }
void display()
 {
        cout << "Count: " << value << endl;
  }
};

int main()
{
    Count count1;
    // Call the "void operator ++ ()" function
    ++count1;
    count1.display();
    return 0;
}
```

Output:

```
Count: 6
```

```
// Overload ++ when used as prefix and postfix
#include <iostream>
using namespace std;
class Count {
   private:
    int value;
   public:
    // Constructor to initialize count to 5
    Count(){
        value = 5;
        }
    // Overload ++ when used as prefix
    void operator ++ () {
        ++value;
    }
// Overload ++ when used as postfix
    void operator ++ (int) {
        value++;
    }
 void display() {
        cout << "Count: " << value << endl;
    }
};

int main() {
    Count count1;
    // Call the "void operator ++ (int)" function
    count1++;
    count1.display();
    // Call the "void operator ++ ()" function
         ++count1;
        count1.display();
        return 0;
}
```

Output:

```
Count: 6
Count: 7
```

```
#include <iostream>
using namespace std;
class Count {
   int value;
   public:
   Count() // Constructor to initialize count to 5
   {
     value = 5;
   }

   Count operator ++ () // Overload ++ when used as prefix
   {
       Count temp;
        // Here, value is the value attribute of the calling object
        temp.value = ++value;
        return temp;
   }
```

```
    // Overload ++ when used as postfix
    Count operator ++ (int)
    {
        Count temp;
        // Here, value is the value attribute of the calling object
        temp.value = value++;
        return temp;
    }
    void display() {
        cout << "Count: " << value << endl;
    }
};
```

```
int main()
{
    Count count1, result;
    // Call the "Count operator ++ ()" function
    result = ++count1;
    result.display();
    // Call the "Count operator ++ (int)" function
    result = count1++;
    result.display();
    return 0;
}
```

Output:

```
Count: 6
Count: 6
```

AMBITION GURU

Example program showing overloading of unary prefix operator(++) using member operator function and unary prefix(--) using friend function.

```
#include<iostream>
using namespace std;
class counter
{
int count;
public:
counter( )
{
  count=0;
}

counter(int c)
{
  count=c;
}

int ret_value( )
{
   return count;
}
void operator ++( )
{
  ++count;
}
friend counter operator --(counter &);
};
```

```
counter operator --(counter &c)
{
counter temp;
temp.count= -- c.count;
return temp;
}
int main( )
{

counter c1(10);


counter c2;
++c1;//c1.operator ++( );
cout<<“The value of c1 is:”<<c1.ret_value( )<<endl;
c2=--c1;//c2=operator--(c1);
cout<<“The value of c1 is:”<<c1.ret_value( )<<endl;
cout<<“The value of c2 object is:”<<c2.ret_value( )<<endl;
 return 0;
}
```

AMBITION GURU

Output:

```
The value of c1 is:11
The value of c1 is:10
The value of c2 object is:10
```

Overloading Binary Operator

- The operator which is used between two operands is called binary operator.
- The binary operator can be overloaded with the help of member operator function as well as non-member operator function(friend function).

AMBITION GURU

Syntax of overloading of binary operator with the help of member operator function:

```
class class_name
{
 //private
data_members;
public:
return_type operator operator_symbol(argument1)
{
  //Body of the member operator function.
}
};
```

// Overloading of the binary operator +. This program adds two complex numbers

```
#include <iostream>
using namespace std;
class Complexx {
    float real,imag;
   public:
   Complexx() // Constructor to initialize real and imag to 0
    {
       real = 0; imag =0;
     }
    void input() {
        cout << "Enter real and imaginary parts respectively: ";
        cin >> real;
        cin >> imag;
     }
```

AMBITION GURU

// Overload the + operator

```
     Complexx operator + (Complexx obj) {
         Complexx temp;
         temp.real = real + obj.real;
         temp.imag = imag + obj.imag;
         return temp;
     }

    void output()
    {
         if (imag < 0)
             cout << "Output Complex number: " << real << imag << "i";
         else
             cout << "Output Complex number: " << real << "+" << imag << "i";
     }
};
```

```
int main() {
    Complexx complex1, complex2, result;
    cout << "Enter first complex number:\n";
    complex1.input();
    cout << "Enter second complex number:\n";
    complex2.input();
   // complex1 calls the operator function
   // complex2 is passed as an argument to the function
    result = complex1 + complex2;
    result.output();
    return 0;
}
```

AMBITION GURU

Output:

```
Enter first complex number:

Enter real and imaginary parts respectively: 1

2

Enter second complex number:

Enter real and imaginary parts respectively: 6

7

Output Complex number: 7+9i
```

AMBITION GURU

Write a C++ program to demonstrate binary operator overloading (+ and -) using member functions for a class named Money that handles values in rupees and paisa.

```
#include<iostream>
using namespace std;
class Money
{
int rupees, paisa;

public:
void input( )
{
 cout<<"Enter the values for rupees and paisa:"<<endl;
 cin>>rupees>>paisa;
}
void display( )
{
 cout<<rupees<<"Rupees and"<<paisa<<"Paisa"<<endl;
}
Money operator +(Money m){
Money temp;
   int s=(paisa+rupees*100)+(m.paisa+m.rupees*100);
   temp.rupees=s/100;
   temp.paisa=s%100;
   return temp;
}
Money operator -(Money m){
   Money temp;
   int d=(paisa+rupees*100)-(m.paisa+m.rupees*100);
   if(d< 0)
       d= -d;
   temp.rupees=d/100;
   temp.paisa=d%100;
   return temp;
}};
```

```
int main( )
{
   Money m1,m2,m3,m4;
   m1.input( );
   m2.input( );
   m1.display( );
   m2.display( );
   m3=m1+m2;//m3=m1.operator+(m2);
   m4=m1-m2;//m4=m1.operator-(m2);
   cout<<"After the addition of two money objects result=";
   m3.display( );
   cout<<"After the subtraction of two money objects result=";
   m4.display( );
return 0;
}
```

Output:

```
Enter the values for rupees and paisa:
1
50
Enter the values for rupees and paisa:
5
60
1Rupees and50Paisa
5Rupees and60Paisa
After the addition of two money objects result=7Rupees and10Paisa
After the subtraction of two money objects result=4Rupees and10Paisa
```

Note: While overloading binary operator through member operator function the left hand side of the operand is used to invoke the member operator function while right hand side of the operand is passed as an argument in the member operator function.

AMBITION GURU

Syntax of binary operator overloading through non-member operator function or friend function:

```
class class_name
{
//private data_members;
public:
friend return_type operator operator_symbol(arg1,arg2);
};

return_type operator operator_symbol(arg1,arg2)
{
//Body of non-member operator function(friend function).
}
```

Example program of binary operator overloading through non-member operator function(friend function)

AMBITION GURU

```
#include<iostream>
using namespace std;
class Distance
{
int feet,inches;
public:
void input( )
{
cout<<"Enter the values in feet and inches:"<<endl;
cin>>feet>>inches;
}
```

```
 void display ( )
{
 cout<<feet<<" Feet and "<<inches<<" Inches"<<endl;
}
 friend void operator +(Distance,Distance);
  friend void operator-(Distance,Distance);
};


 void operator +(Distance d1,Distance d2)
{
 int s=(d1.inches+d1.feet*12)+(d2.inches+d2.feet*12);
  int f=s/12;
 int i= s%12;
 cout<<f<<" Feet and"<<i<<" Inches"<<endl;
}
```

```
void operator -(Distance d1,Distance d2)
{
int d=(d1.inches+d1.feet*12)-(d2.inches+d2.feet*12);
if(d<0)
d= -d;
int f=d/12;
int i= d%12;
cout<<f<<" Feet  and"<<i<<" Inches"<<endl;
}
```

```
int main( )
{
Distance d1, d2;
d1.input( );
d2.input( );
d1.display( );
d2.display( );
cout<<"After the addition of two distance objects result="<<endl;
d1+d2; //operator+(d1,d2)
cout<<"After the subtraction of two distance objects result="<<endl;
d1-d2; // operator-(d1,d2)
return 0;
}
```

Output:

```
Enter the values in feet and inches:
1
11
Enter the values in feet and inches:
2
10
1 Feet   and 11 Inches
2 Feet   and 10 Inches
After the addition of two distance objects result=
4 Feet   and 9Inches
After the subtraction of two distance objects result=
0 Feet and 11Inches
```

Note:

- In case of binary operator overloading through non-member operator function(friend function) both the left and right hand side of the operand are passed as an argument in the non-member operator function or friend function.

Mandatory use of friend function while overloading binary operators:

Let us consider the statements, c2=c1+5;

c2=5+c1;

Where, c1 and c2 are objects of type complexx.

- The first statement c2=c1+5 can be overloaded by help of both member operator function as well as friend function however the statement c2=5+c1 can only be overloaded by the help of a friend function.
- This is because we know that in case of binary operator overloading by the help of member operator function the left hand side of the operand is used to invoke the member operator function which should be the object of a class.

- But in the statement c2=5+c1 the left hand side of the operand is built in data type so it cannot be used to invoke the member operator function.
- So in this case the friend function becomes mandatory as friend function is invoked as the normal function and both left hand side operand and right hand side operand are passed as an arguments.

Example

WAP to achive operations like c2=c1+5 and c2=5+c1 where c1 and c2 are objects of type complexx.

```
 #include<iostream>
 using namespace std;
 class complexx
{
 float real,imag;
 public:
 void input( ){
   cout<<"Enter the real and imaginary part:"<<endl;
   cin>>real>>imag;
}
 void display( ){
   cout<<"("<<real<<" , "<<imag<<")"<<endl;
}
complexx operator +( int p){
    complexx temp;
    temp.real=real+p;
    temp.imag=imag;
     return temp;
}
friend complexx operator+(int,complexx);
};

complexx operator+(int p,complexx c){
complexx temp;
temp.real=p+c.real;
temp.imag=c.imag;  return temp;
}
```

```
int main( )
{
complexx c1,c2;
c1.input( );

c1.display ( );

c2=c1+5; //c2=c1.operator+(5);
cout<<"Addition using member operator function the result=";

c2.display( );

c2=5+c1; //c2=operator+(5,c1);

cout<<"Using non-member operator function(friend function) the result=";

c2.display( );

 return 0;
 }
```

Output:

```
Enter the real and imaginary part:
3
4
(3 , 4)
Addition using member operator function the result=(8 , 4)
Addition using non-member operator function(friend  function) the result=(8 , 4)
```

AMBITION GURU

Inheritance: Derived Class & Base Class: Specifying the Derived class accessing Base class members, the protected access specifier, Derived class constructor, Overriding member functions, public and private inheritance; Access Combinations, Classes & Structures, Access Specifiers. Multilevel inheritance, Hybrid inheritance, Multiple inheritance; member functions in multiple inheritance, constructors in multiple inheritance, Containership; Classes, within classes, Inheritance & Program development.

Introduction to inheritance

C++ supports the concept of reusability.

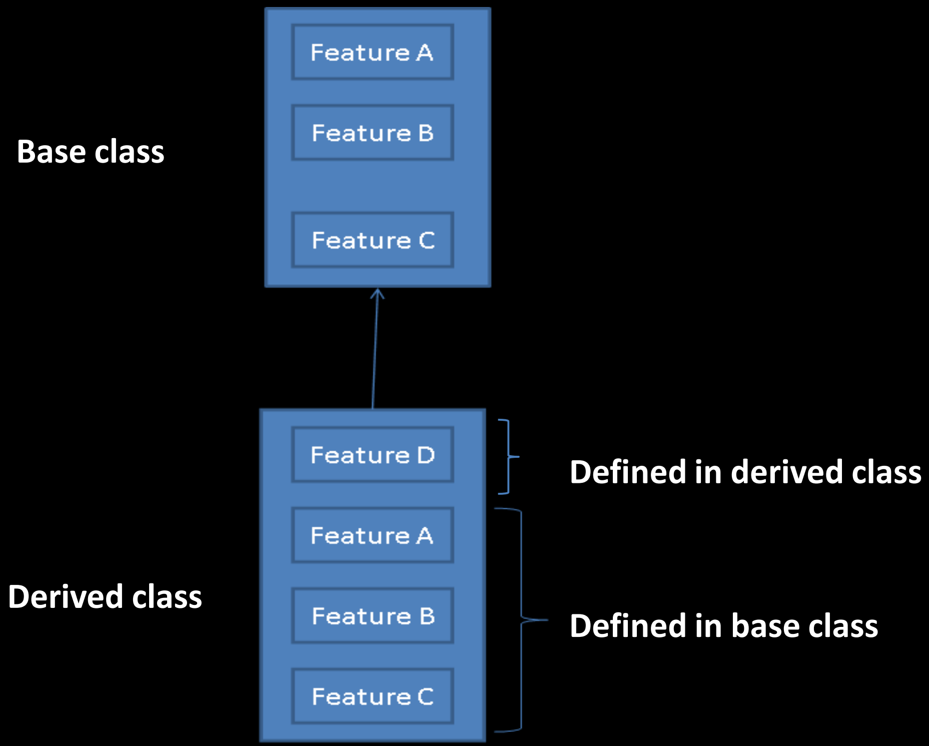
C++ classes can be reused repeatedly by the programmers to suit their requirements.

This can be achieved by creating new classes from the existing one. ***The process of creating new class from the existing one is called inheritance.***

Base and Derived class

The existing class is called an ***ancestor class*** or ***base class*** or ***parent class*** or ***superclass***.

And the new class created from the existing one is called a ***descendant class*** or ***derived class*** or ***child class*** or ***sub class***.


Base class
Feature A
Feature B
Feature C
Derived class
Feature D
Feature A
Feature B
Feature C
Defined in derived class
Defined in base class

AMBITION GURU

C++ example showing base and derived class concepts of Vehicle → Car relationship

AMBITION GURU

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

// Base class
class Vehicle {
protected:
    string brand;
    int year;

public:
    Vehicle(string b, int y) {
        brand = b;
        year = y;
    }

    void showInfo() {
        cout << "Brand: " << brand << ", Year: " << year << endl;
    }
};
```

```cpp
// Derived class
class Car : public Vehicle {
private:
    string model;

public:
    Car(string b, int y, string m) : Vehicle(b, y) {
        model = m;
    }

    void showCarInfo() {
        showInfo();
        cout << "Model: " << model << endl;
    }
};
```

```
int main() {
    Car myCar("Toyota", 2022, "Corolla");
    myCar.showCarInfo();

    return 0;
}
```

Output:

```
Brand: Toyota, Year: 2022
Model: Corolla
```

Access Specifiers

AMBITION GURU

- **C++ access specifiers** are used for determining or setting the boundary for the availability of class members (data members and member functions) beyond that class.
- Access modifiers (or access specifiers) are keywords in object-oriented languages that set the accessibility of classes, methods, and other members.
- Access modifiers are a specific part of programming language syntax used to facilitate the encapsulation of components.
- Following are the access specifiers available in C++
  - ➢ private access specifier
  - ➢ protected access specifier
  - ➢ public access specifier

- By default, functions and data declared within a class are private to that class and may be accessed only by other members of the class.
- The **public** access specifier allows functions or data to be accessible to other parts of our program.
- The **protected** access specifier is needed only when inheritance is involved

AMBITION GURU

Protected access specifier and its role in inheritance

The private and protected access specifiers has same effect in a class however their difference is seen in case of inheritance only.

The ***protected*** members are ***visible*** to ***derived class*** however ***private*** members ***are not***.

The purpose of making data member (s) as protected in a class is to make such members accessible to derived class functions.

Syntax of derived class declaration:

```
class Derived_class_name : visibility mode Base_class_name
{
        //Body of derived class
}
```

- The colon(:) after the Derived_class_name indicates that the new class is created from the existing one.
- Here visibility mode specifies either public or private or protected.

Visibility mode and its effect in inheritance

- Only the public and protected members of a class gets inherited in a derived class while private members of a class are not inherited.

Visibility Mode	Inheritable public members of a baseclass in a derived class becomes	Inheritable protected membersof a base class in a derived class becomes	Private members ofa base class are not inherited in a derived class
public	public	protected	
protected	protected	protected	
private	**private**	**private**	

AMBITION GURU

Data Member Overriding

AMBITION GURU

Data member overriding

- If the base class and derived class have the same name of the *data member(s)* then it is called ***data member overriding***
- The data member(s) of the base class are hidden in the derived class i.e., data member of derived class overrides (hides or displaces) the data member(s) of the base class.

Member function overriding

- If the base class and derived class have the same name of the member function, then it is called ***member function overriding***
- The member function of the base class is hidden in the derived class i.e. member function of derived class overrides(hides or displaces) the member function of the base class.

```
#include<iostream>
using namespace std;
class Base{
  protected:
    int n;
  public:
    void input(){
     cout<<"Enter the value to the n of a base class:"<<endl;
     cin>>n;
}
void display(){
     cout<<"The value to the base of n="<<n<<endl;
}
};
```

```
class Derived:public Base{
   protected:
     int n;
   public:
      void input( ){
     cout<<"Enter the value to the n of a derived class:"<<endl;
     cin>>n;
}
void display( )
{
     cout<<"The value to the n of a derived class is:"<<n<<endl;
     cout<<"The value of both the n is :"<<n+n<<endl;
}
};
```

```
int main( )
{
    Derived d;
    d.input(); //with the intention of calling Base input()
    d.display( ); //with the intention of calling Base display()
    d.input( ); //with the intention of calling Derived input()
    d.display( ); //with the intention of calling Derived display()
    return 0;
}
```

Output:

```
Enter the value to the n of a derived class:
3
The value to the n of a derived class is: 3
The value of both the n is :6
Enter the value to the n of a derived class:
2
The value to the n of a derived class is: 2
The value of both the n is :4
```

Overridden members of a base class can be invoked by two ways:

Overridden members of a base class can be invoked by twoways:

1. By the help of a member function of a derived class.
2. By the help of a derived class object.

In both the cases class_name scope resolution(::) and the name ofthe member is written.

By the help of a derived class function:

```
#include<iostream>
using namespace std;
class Base
{
protected:
int n;
public:
```

```
void input( )
{
    cout<<"Enter the value to the n of a base class:"<<endl;
    cin>>n;
}

void display( )
{
    cout<<"The value to the n of a base class="<<n<<endl;
}
};

class Derived:public Base
{
protected:
int n;
public:
```

```
void input( )
{
 Base::input( );
 cout<<"Enter the value to the n of a derived class:"<<endl;
 cin>>n;
}

void display( )
{
 Base::display( );
 cout<<"The value to the n of derived class is:"<<n<<endl;
 cout<<"The value of both the n is:"<<n+Base::n;
}};
```

```
int main()
{
    Derived d;
    d.input();
    d.display();
    return 0;
}
```

Output:

```
Enter the value to the n of a base class:
3
Enter the value to the n of a derived class:
5
The value to the n of a base class=3
The value to the n of derived class is:5
The value of both the n is:8
```

By the help of a derived class object:

```
#include<iostream>
using namespace std;
class Base{
protected:
int n;
public:
void input( ){
cout<<"Enter the value to the n of a base class:"<<endl;
cin>>n;
}
void display( ){
cout<<"The value to the n of a base class="<<n<<endl;
}
};
```

```
class Derived:public Base{
protected:
int n;
public:
void input( )
{
     cout<<"Enter the value to the n of a derived class:"<<endl;
     cin>>n;
}

void display( )
{
     cout<<"The value of n of a derived class is:"<<n<<endl;
     cout<<"The value of both the n is:"<<n+Base::n;
}};
```

```
int main( )
{
    Derived d;
    d.Base::input( );
    d.Base::display( );
    d.input( );  //invokes derived class input( );
    d.display( ); //invokes derived class display( );
    return 0;
}
```

Output:

```
Enter the value to the n of a base class:
3
The value to the n of a base class=3
Enter the value to the n of a derived class:
2
The value of n of a derived class is:2
The value of both the n is:5
```

Types of inheritance

Types of inheritance

1. Single Inheritance
2. Multiple Inheritance
3. Multilevel Inheritance
4. Hierarchical Inheritance

1. Single Inheritance

The type of inheritance in which there is a ***single base class, and a single derived*** class is called single inheritance.

AMBITION GURU

The general form(syntax):

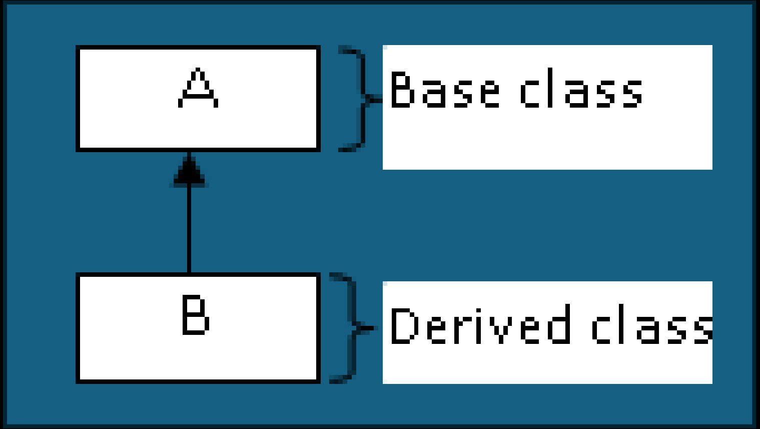
```
class base
{………..};

class derived: visibility_mode base
{………..};
```

For Example:

```
class A
{…};

class B:public A
{…};
```



//Example Program:

AMBITION GURU

```
#include<iostream>
using namespace std;
class employee{
protected:
    char name[20];
    float salary;
public:
void input( )
{
    cout<<"Enter the name of an employee:"<<endl;
    cin>>name;
    cout<<"Enter the salary of an employee:"<<endl;
    cin>>salary;
}
```

```
void display( ){
     cout<<"Name of an employee="<<name<<endl;
     cout<<"Salary of an employee="<<salary<<endl;
}};

class manager:public employee
{
protected:
     char title[20];
public:
void input( )
{
     employee::input( );
     cout<<"Enter the title of an manager:"<<endl;
     cin>>title;
}
```

```
void display( ){
    employee::display( );
    cout<<"The title of a manager="<<title<<endl;
}};

int main( )
{
    manager m1,m2;
    m1.input( );
    m2.input( );
    m1.display( );
    m2.display( );
    return 0;
}
```

AMBITION GURU

Output:

```
Enter the name of an employee:
Mark
Enter the salary of an employee:
560
Enter the title of an manager:
Engineer
Enter the name of an employee:
Samson
Enter the salary of an employee:
450
Enter the title of an manager:
Supervisor
```

```
Name of an employee = Mark
Salary of an employee = 560
The title of a manager = Engineer
Name of an employee = Samson
Salary of an employee = 450
The title of a manager = Supervisor
```

2. Multiple Inheritance:

The type of inheritance in which there are multiple(more than one) base classes, and a single derived class is called multiple inheritance.

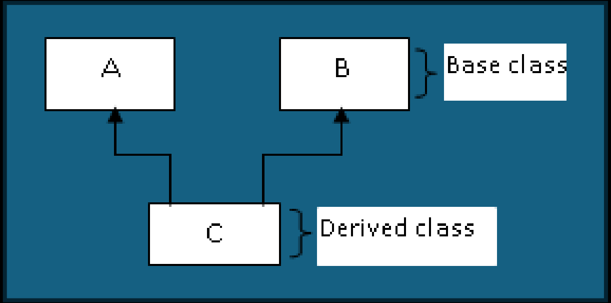
- The syntax of multiple inheritance is :

```
class base1
{……….};
class base2
{……….};

.

.
class basen
{…………};

class derived:visibility_mode base1,visibility_mode base2,..
{………….};
```



For Example:

```
class A
{…};

class B
{…};

class C:public A, public B
{…};
```

//Example program:

AMBITION GURU

```
#include<iostream>
using namespace std;
class Base1
{
protected:
int x;
public:

void get_x( )
{
     cout<<"Enter the value to x of Base1:"<<endl;
     cin>>x;
}

void display_x( )
{
     cout<<"The value to x of Base1="<<x<<endl;
}
};
```

```
class Base2
{
    protected:
    int y;
    public:

void get_y( )
{
    cout<<"Enter the value to y of a Base2:"<<endl;
    cin>>y;
}

void display_y( )
{
    cout<<"The value to the y of a Base2="<<y<<endl;
}
};
```

```
class derived: public Base1,public Base2
{
protected:
int z;

public:
void get_z( )
{
     cout<<"Enter the value to z of a derived class:"<<endl;
     cin>>z;
}

void display_z( )
{
     cout<<"The value to the z of a derived class="<<z<<endl;
}

void display( )
{
     cout<<"The sum is:"<<x+y+z<<endl;
}
};
```

```
int main( )
{
     derived d;
     d.get_x( );
     d.get_y( );
     d.get_z( );
     d.display_x( );
     d.display_y( );
     d.display_z( );
     d.display( );
     return 0;
}
```

Output:

```
Enter the value to x of Base1:
4
Enter the value to y of a Base2:
5
Enter the value to z of a derived class:
6
The value to x of Base1=4
The value to the y of a Base2=5
The value to the z of a derived class=6
The sum is:15
```

Ambiguity and its resolution in multiple inheritance:

In multiple inheritance there are multiple base classes and a single derived class.

If multiple base classes have same members name which are being inherited to derived class and when we access those members by using the object of a derived class, then the compiler becomes confused to use which version of member and of which base class and this is called the ambiguity in multiple inheritance.

In order to resolve this problem while invoking the same members name which are in more than one base classes, we use class name and scope resolution operator while invoking those members.

```
#include<iostream>
using namespace std;
class Base1
{
    protected:
    int x;
    public:
    void get_x( )
    {
    cout<<"Enter the value to x of a Base1:"<<endl;
    cin>>x;
    }

    void display_x( )
    {
    cout<<"The value to x of  Base1="<<x<<endl;
    }
};
```

```
class Base2
{
    protected:
    int x;
    public:
    void get_x( )
    {
        cout<<"Enter the value to x of a Base2:"<<endl;
        cin>>x;
    }

    void display_x( )
    {
        cout<<"The value to x of a Base2="<<x<<endl;
    }
};
```

```
class derived:public Base1,public Base2
{
protected:
int z;
public:
void get_z( )
{
     cout<<"Enter the value to z of a derived class:"<<endl;
     cin>>z;
}

void display_z( )
{
     cout<<"The value to the z of a derived class="<<z<<endl;
}

void display( )
{
     cout<<"The sum is:"<<Base1::x+Base2::x+z<<endl;
}
};
```

```
int main( )
{
derived d;
//d.get_x( );error
d.Base1::get_x( );
//d.display_x( );error
d.Base1::display_x( );

d.Base2::get_x( );
d.Base2::display_x( );
d.get_z( );
d.display_z( );
d.display( );
return 0;
}
```

Output:

```
Enter the value to x of Base1:
4
Enter the value to x of a Base2:
5
Enter the value to z of a derived class:
6
The value to x of Base1=4
The value to the x of a Base2=5
The value to the z of a derived class=6
The sum is:15
```

3. Multilevel Inheritance:

The type of inheritance in which derived class is formed from already derived class is called multilevel inheritance. The transitive nature of inheritance is shown by multilevel inheritance.

3. Multilevel Inheritance:

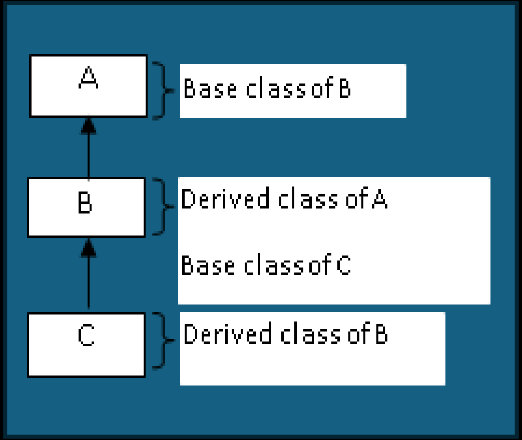
The type of inheritance in which derived class is formed from already derived class is called multilevel inheritance. The transitive nature of inheritance is shown by multilevel inheritance.

General form(syntax):

```
class base
{……..};

class derived1:visibility_mode base
{………..};

class derived2:visibility_mode derived1
{………….};
```



AMBITION GURU

For Example:

```
class A
{…};

class B:public A
{…};

class C:public B
{…};
```

//Example Program

AMBITION GURU

```
#include<iostream>
using namespace std;
class student
{
protected:
     char name[20];
     int roll;
public:
void input( )
{
     cout<<"Enter the name of an student:"<<endl;
     cin>>name;
     cout<<"Enter the roll number of an student:"<<endl;
     cin>>roll;
}
```

```
void display( )
{
     cout<<"The name of an student="<<name<<endl;
     cout<<"The roll number of an student="<<roll<<endl;
}
};

class test:public student
{
protected:
     float sub1,sub2;
public:
void input( )
{
     student::input( );
     cout<<"Enter the marks in subject1 and subject2:"<<endl;
     cin>>sub1>>sub2;
}
```

```
void display( )
{
     student::display( );
     cout<<"The marks of subject1="<<sub1<<endl;
     cout<<"The marks of subject2="<<sub2<<endl;
}
};

class result: public test
{
protected:
     float total; public:
void input( )
{
     test::input( );
     total=sub1+sub2;
}
```

```
void display( )
{
    test::display( );
    cout<<"Total="<<total<<endl;
}
};

int main( )
{
    result r;
    r.input( );
    r.display( );
    return 0;
}
```

Output:

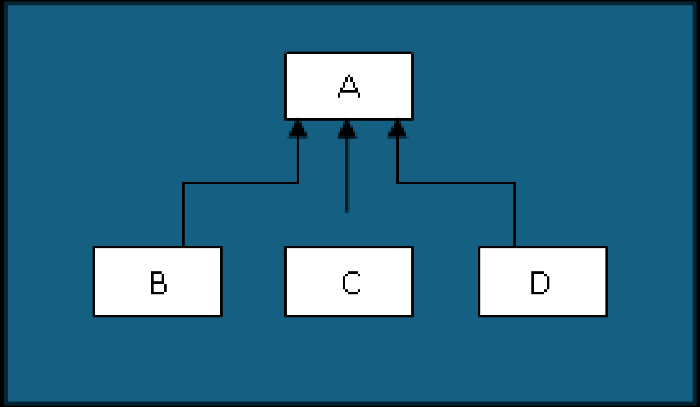
```
Enter the name of an student:
Ramesh
Enter the roll number of an student:
34
Enter the marks in subject1 and subject2:
3
4
The name of an student=Ramesh
The roll number of an student=34
The marks of subject1=3
The marks of subject2=4
Total=7
```

4. Hierarchical Inheritance

The type of inheritance in which there is a single base class, and multiple derived classes is called hierarchical inheritance.

The general form is:

```
class base
{…………};
class derived1 : visibility_mode base
{        …..     };
class derived2 : visibility_mode base
{        ….      };
class derived3 : visibility_mode base
{        ….      };
.
.
class derivedn:visibility_mode base
{        ….      };
```



AMBITION GURU

For Example:

```
class A
{…};

class B : public A
{…};

class C : public A
{…};

class D : public A
{…};
```

//Example Program:

AMBITION GURU

```
#include<iostream>
using namespace std;
class employee{
protected:
    char name[20];
    float salary;
public:
void input( ){
    cout<<"Enter the name of an employee:"<<endl;
    cin>>name;
    cout<<"Enter the salary of an employee:"<<endl;
    cin>>salary; }

void display( ){
    cout<<"Name of an employee="<<name<<endl;
    cout<<"Salary of an employee="<<salary<<endl;
}
};
```

```
class manager: public employee
{
protected:
    char title[20];
public:
void input( )
{
    employee::input( );
    cout<<"Enter the title of a manager:"<<endl;
    cin>>title;
}
void display( )
{
    employee::display( );
    cout<<"The title of a manager="<<title<<endl;
}
};
```

```
class teacher:public employee
{
protected:
     char faculty[20];
public:
void input( )
{
     employee::input( );
     cout<<"Enter the faculty of the teacher:"<<endl;
cin>>faculty;
}
void display( )
{
     employee::display( );
     cout<<"The faculty of the teacher="<<faculty;
}
};
```

```
int main( )
{
     cout<<"Manager:"<<endl;
     manager m;
     m.input( );
     cout<<"Teacher:"<<endl;
     teacher t;
     t.input( );
     cout<<"Manager details:"<<endl;
     m.display( );
     cout<<"Teacher details:"<<endl;
     t.display( );
     return 0;
}
```

Output:

```
Manager:
Enter the name of an employee:
Vicky
Enter the salary of an employee:
450
Enter the title of a manager:
Supervisor
Teacher:
Enter the name of an employee:
Kaushal
Enter the salary of an employee:
560
Enter the faculty of the teacher:
CSIT
```

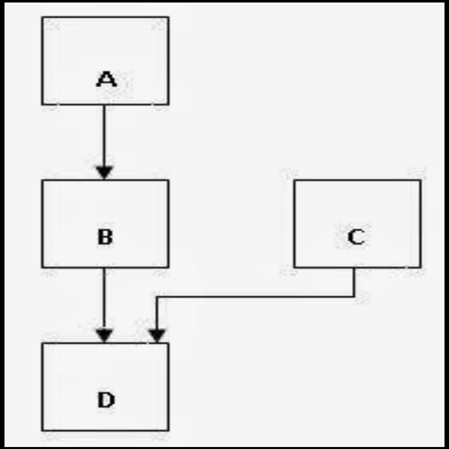
```
Manager details:
Name of an employee=Vicky
Salary of an employee=450
The title of a manager=Supervisor
Teacher details:
Name of an employee=Kaushal
Salary of an employee=560
The faculty of the teacher=CSIT
```

Hybrid Inheritance:

The type of inheritance in which there are more than one inheritances mixed together.

General form (syntax):

```
class base1
{…………..};
class derived1:visibility_mode base1
{…………..};
class base2
{……………};
class derived2:visibility_mode- derived1,visibility_mode base2
{……………};
```


$$\begin{tikzcd}
A \arrow[d] & \\
B \arrow[d] & C \arrow[dl] \\
D &
\end{tikzcd}$$

Hybrid Inheritance

Example:

```
class A{…};

class B:public A

{…};

class C{…};

class D: public B, public C

{…};
```

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

class student
{
protected:
    char name[20];
    int roll;
public:
void input( )
{
    cout<<"Enter the name of the student:"<<endl;
    cin>>name;
    cout<<"Enter the roll number of the student:"<<endl;
    cin>>roll;
}
```

```
void display( )
{
     cout<<"Name of the student="<<name<<endl;
     cout<<"Roll number of the student="<<roll<<endl;
}
};

class test: public student
{
protected:
     float sub1,sub2;
public:
void input( )
{
     student::input( );
     cout<<"Enter the marks in subject1 and subject2:"<<endl;
     cin>>sub1>>sub2;
}
```

```
void display( )
{
    student::display( );
    cout<<"The marks of subject1="<<sub1<<endl;
    cout<<"The marks of subject2="<<sub2<<endl;
}
};

class sports
{
protected:
    float sm;
public:
void input( )
{
    cout<<"Enter the marks of sports:"<<endl;
    cin>>sm;
}
```

```
void display( )
{
     cout<<"The marks of the sports="<<sm<<endl;
}
};

class result: public test, public sports
{
protected:
     float total;
public:
void input( )
{
     test::input( );
     sports::input( );
     total=sub1+sub2+sm;
}
```

```
void display( )
{
     test::display( );
     sports::display( );
     cout<<"Total marks="<<total<<endl;
}
};

int main( )
{
     result r;
     r.input( );
     r.display( );
      return 0;
}
```

Output:

```
Enter the name of the student:
Arjun
Enter the roll number of the student:
34
Enter the marks in subject1 and subject2:
23
45
Enter the marks of sports:
50
Name of the student=Arjun
Roll number of the student=34
The marks of subject1=23
The marks of subject2=45
The marks of the sports=50
Total marks=118
```

AMBITION GURU

Constructors and destructors in inheritance

Constructor in derived class:

- If there is no constructor at all or there is a default constructor in a base class then it is not necessary to have a constructor in a derived class.
- However if there is a parameterized constructor in a base class then it is mandatory to have constructor in a derived class too because it is the job of derived class constructor to pass value(s) to the base class constructor.

```
//Example program:
#include<iostream>
using namespace std;
class alpha
{
protected:
    int x;
public:
alpha(int i)
{
 x=i;
 cout<<"alpha initialized"<<endl;
}
void display_x( )
{
 cout<<"x="<<x<<endl;
}
};
```

```
class beta
{
protected:
    int y;
public:
beta( int j)
{
    y=j;
    cout<<"beta initialized"<<endl;
}
void display_y( )
{
     cout<<"y="<<y<<endl;
}};
class gamma: public alpha, public beta
{
protected:
     int m,n;
public:
gamma(int a,int b,int c,int d):alpha(a),beta(b)
{
     m=c;
     n=d;
     cout<<"gamma initialized"<<endl;
}};
```

```
void display_mn( )
{
    cout<<"m="<<m<<endl;
    cout<<"n="<<n<<endl;
}
};
int main( )
{
gamma g(11,12,13,14);
g.display_x( );
g.display_y( );
g.display_mn( );
return 0;
}
```

Output:

```
alpha initialized
beta initialized
gamma initialized
x=11
y=12
m=13
n=14
```

AMBITION GURU

Constructor and destructor invocation order in case of single inheritance

Constructor and destructor invocation order in case of single inheritance

AMBITION GURU

- If there are constructors in base class and derived class in single inheritance, then first the constructor of base class is invoked and after that the constructor of derived class is invoked.
- The destructors invocation order is the reverse order of invocation of constructor.

```
#include<iostream>
using namespace std;
class Base
{
public:
Base( )
{
cout<<"Base class constructor"<<endl;
}
~Base( )
{
    cout<<"Base class destructor"<<endl;
}
};

class Derived:public Base
{
public:
Derived( )
{
    cout<<"Derived class constructor"<<endl;
}
~Derived( )
{
    cout<<"Derived class destructor"<<endl;
}};
```

```
int main( )
{
{
Derived d;
}
    cout<<"End of main"<<endl;
return 0;
}
```

Output:

```
Base class constructor
Derived class constructor
Derived class destructor
Base class destructor
End of main
```

AMBITION GURU

Constructor and destructor invocation order in case of multiple inheritance

Constructor and destructor invocation order in case of multiple inheritance

- The constructor's invocation in multiple inheritance take places according to the order of the derived class declaration.
- The destructors invocation order is reverse to the order of invocation of the constructor.

```
#include<iostream>
using namespace std;
class Base1
{
public:
Base1( )
{
    cout<<"Constructor from Base1:"<<endl;
}
~Base1( )
{
    cout<<"Destructor from Base1:"<<endl;
}};

class Base2
{
public:
Base2( )
{
    cout<<"Constructor from Base2:"<<endl;
}
~Base2( )
{
    cout<<"Destructor from Base2:"<<endl;
}};
```

```
class Derived:public Base1,public Base2
{
public:
Derived( )
{
    cout<<"Constructor from Derived:"<<endl;
}
~Derived( )
{
    cout<<"Destructor from Derived:"<<endl;
}
};
```

```
int main( )
{
{
Derived d;
}
    cout<<"End of main:"<<endl;
return 0;
}
```

Output:

```
Constructor from Base1:
Constructor from Base2:
Constructor from Derived:
Destructor from Derived:
Destructor from Base2:
Destructor from Base1:
End of main:
```

Constructor and destructor invocation order in case of multilevel inheritance

Constructor and destructor invocation order in case of multilevel inheritance

- The order of constructor's invocation takes place in case of multilevel inheritance according to the order of inheritance.
- Destructors invocation takes place according to the reverse order of the invocation of the constructor.

```
//Example program
#include<iostream>
using namespace std;
class Base
{
public:
Base( ){
    cout<<"Base class constructor:"<<endl;
}
```

```
~Base( )
{
    cout<<"Base class destructor:"<<endl;
}
};

class derived1:public Base
{
public:
derived1( )
{
    cout<<"derived1 class constructor:"<<endl;
}
```

```
~derived1( )
{
    cout<<"derived1 class destructor:"<<endl;
}
};

class derived2:public derived1
{
public:
derived2( )
{
    cout<<"derived2 class constructor:"<<endl;
}
```

```
~derived2( )
{
    cout<<"derived2 class destructor:"<<endl;
}
};

int main( )
{
{
derived2 d;
}
    cout<<"End of main:"<<endl;
return 0;
}
```

Output:

```
Base class constructor:
derived1 class constructor:
derived2 class constructor:
derived2 class destructor:
derived1 class destructor:
Base class destructor:
End of main:
```

Containership

AMBITION GURU

- One class contains another class as a member.
- Also called composition. Represents a "has-a" relationship.

Example: A Car has-a Engine.

```
class Engine {
public:
    void start()
     {
          cout << "Engine Started\n";
     }
};

class Car {
    Engine e;    // Car contains Engine
public:
    void drive() {
        e.start();
        cout << "Car is moving\n";
    }
};
```

Classes within Classes (Nested Classes)

AMBITION GURU

- A class defined inside another class.
- The inner class is usually used only by the outer class.

Example: A Car has-a Engine.

```
class University {
public:
    class Department {    // Nested class
    public:
        void show() { cout << "Department of 
Computer Science\n"; }
    };
};
```

Calling from main():

```
    University::Department d;
     d.show();
```

Program Development with OOP Concepts

AMBITION GURU

Steps in building a program using classes:

1. **Identify objects** → (e.g., Student, Course, Teacher).
2. **Decide relationships** → (Student *has-a* Course, Student *is-a* Person).
3. **Create class definitions** → Data members + methods.
4. **Use inheritance** → For shared properties.
5. **Use containership** → For “has-a” relationships.
6. **Test and extend** → Reuse existing code.

AMBITION GURU

Questions on operator overloading

1. Write a program to create a class named height with data members meter(int) and centimeter(int). Overload the binary + operator using friend function to add two heights.
2. Explain about friend function and friend class with an example.
3. Create a class named Point with data members x(int) and y(int). Add operator overloading to find the Euclidean distance between two points.
4. What is operator overloading? Why it is necessary to overload an operator? Write a program for overloading comparison operator.
5. What is the concept of friend function? How it violates the data hiding principle? Justify with example.
6. Define operator overloading. Write rules for overloading operator. WAP to concatenate two strings using binary operator overloading.

Questions on inheritance

AMBITION GURU

1. Explain the practical implication of protected specifier in inheritance. List advantages and disadvantages of inheritance.
2. Define inheritance. List down its pros and cons.
3. What is overriding? Write a program for implementing following: Create a class author with attributes name and qualification. Also create a class publication with pname. From these classes derive a classes derive a class book having attributes title and price. Each of the three classes should have getdata() method to get their data from user. The classes should have putdata() method to display the data. Create instance of the class book in main.
4. What are the various class access specifiers? How public inheritance differs from private inheritance?
5. How ambiguity occurs in multiple inheritances? Explain with an example how ambiguity can be resolved?

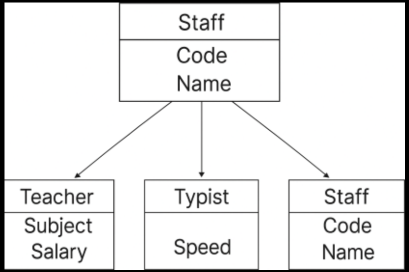
Questions on inheritance

6. Explain the working of multiple and multilevel inheritance in brief.

7. Define derived class and base class. Explain any two types of inheritance in brief with example.

8. Depict the difference between private and public derivation. Explain derived class constructor with suitable program.

9. State the use of new operator. An educational institute wishes to maintain a data of its employee. The hierarchical relationships of related classes are as follows. Define all the classes to represent below hierarchy and define functions to retrieve individual information as and when required.



10. Explain single and multiple inheritance with their syntax and example.

AMBITION GURU

THANK YOU

Any Queries ?