Static Member function: The member functions of the class can also be made static which are known as static member functions. Some of the features/characteristics of such function are as follows:

- It is declared with the keyword static.
- It is also known as class member function.
- It can be invoked(called) with or without object
- It can only access static members of the class.

Example:

```
class Student{
      int x;
      static int y;    //static member variable

      static void doSomething( ) {     //static member function
        y=y+10;   //accessing static member variable
        x=x+1;    //error //cannot access other members which are not static
      }
};
```

An example of a use of a static member function is shown below. *[Example program- 23]*

Example program- 23:

```
class Model{
  int x;
  static int y; //static member variable declaration
  public:
  static void getSomething(int a){ //static member function
   y=a;   //accessing static member variable
  }
  void displaySomething(){
   cout<<"The value of y: "<<y<<endl;
  }
};
int Model::y;  //static member variable defintion
int main(){
  Model paul, samikshya;
  Model::getSomething(5);//calling static member function without object
  paul.displaySomething();
  paul.getSomething(10); //calling static member function using object
//all objects have the same copy of y, which is a static member variable
  samikshya.displaySomething();
  return 0;
}
```

Sapkota, U-TEC

Output: The value of y: 5
The value of y: 10

Explanation: In the above program, getSomething( ) function is declared as static so that it can access only the static data member (y) and cannot access the member x. Now, when this function is called, it can be called directly using class name 'Model' without the need of objects. If this static function modifies any static data member (for e.g. y in above program), all the objects will be sharing the same copy of that data member as shown in above program.

Example program-24:

Q. Illustrate an example how the destructor is used to release dynamically allocated memory and avoid memory leakage.

```
class Utech{
 int *p;

 public:
 Utech(){   //Constructor
  p= new int; //Allocating memory dynamically
  cout<<"Memory allocated dynamically."<<endl;
 }

 ~Utech(){  //Destructor
   delete p;    //Releasing memory pointed by 'p'
   cout<<"Memory is released."<<endl;
  }
};

int main()
{
  Utech u1;
  return 0;
}
```

Output: Memory allocated dynamically.
Memory is released.

Explanation: In the above program, when an object 'u1' is created, a constructor is called automatically that allocates a memory dynamically for storing an integer. The memory address is stored in the pointer variable 'p'. And before the object goes out of scope, a destructor is called automatically that deletes the pointer 'p'. By deleting the pointer 'p', the memory pointed by it is released, and thus avoids memory leakage.

friend function: It is a special function that can access any members of the class to which it is declared as a friend.

- It is declared in the class with 'friend' keyword.
- It is not a member function of the class to which it is a friend.
- It must be defined outside the class.
- It cannot directly access the members of the class.
- It uses objects to access the members of the class.

An example of a use of a friend function is given below. *[Example program 25]*

Example program- 25:

```
class Nepal{
  int x;
  public:
  Nepal(){    //Default constructor
   x=2;
  }
  friend void happy(Nepal);    //friend function declaration
};

void happy(Nepal n){           //friend function definition
 cout<<"Value: "<<n.x<<endl;
}

int main()
{
   Nepal n1;
   happy(n1);
   return 0;
}
```

Output: Value: 2

Explanation of the program: An outsider function 'happy( )' which is not a member of the class 'Nepal' is declared as a friend inside the class 'Nepal'. So, now the 'happy( )' function can access the data member of that class (in this case x).

In the above program, object 'n1' is passed as argument in the happy( ) function. And the function uses this object to access its member 'x'.

Use cases of friend function:

- It can become friend to more than one class. So, a friend function can be used to access the data members of more than one class at the same time.
- It can be used to overload the operators.

Example program- 26:

Q. Write a program to add any two private numbers of two different classes using friend function.

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

class B;    //declaring 'B' as a class
class A{
  int num1;
  public:
  A(int x){
    num1=x;
  }
  friend int add(A, B); /*It takes objects of both A
                        and B classes as arguments */
};
class B{
  int num2;
  public:
  B(int y){
    num2=y;
  }
  friend int add(A, B);
};

int add(A x, B y){
 int sum=x.num1 + y.num2;
 return sum;
}
int main()
{
    A obj1(2);
    B obj2(5);

    int sum= add(obj1,obj2);
    cout<<"Addition of two numbers: "<<sum<<endl;

    return 0;
}
```

Output: Addition of two numbers: 7

Q. The concept of friend function is against the philosophy of OOP. Justify it.
Answer: One of the important characteristics of OOP is data encapsulation. That means, the data and functions are wrapped into a single unit. The data are kept private and are not accessible to the outside world. These data are accessed by only the member functions that are present in the same unit (i.e. in the same class). This process is also known as data hiding as the data are hidden from outside.

But, there is a special function known as friend function which violates the above principle of data hiding. If any non-member function is declared as a friend inside a class, then such function can access all the private data members of the class.

It can be demonstrated by the following program example.

```
class Nepal{
  int x;
  public:
  Nepal(){    //Default constructor
   x=2;
  }
  friend void happy(Nepal);    //friend function declaration
};

void happy(Nepal n){           //friend function definition
 cout<<"Value: "<<n.x<<endl;
}

int main()
{
   Nepal n1;
   happy(n1);
   return 0;
}
```

Output: Value: 2

In above program, 'x' is a private data member of class 'Nepal'. A non-member function happy( ) is declared as a 'friend' inside the same class. This function then accesses the private data member 'x' using the object of class 'Nepal'. Here, the principle of data hiding and encapsulation is violated. Therefore, the concept of friend function is against the philosophy of OOP.

Inheritance: It is a process of inheriting the properties and behaviors of existing class into a new class. The existing class is known as the parent class or base class or super class. The new class is known as the child class or derived class or sub class. The derived class inherits the features (data and member functions) from the base class, and can have additional features of its own.

Syntax:

```
class Base_class
{
   // data members and member functions
};

class Derived_class : visibility_mode  Base_class
{
  // data members and member functions
};
```

- The colon (:) indicates that the Derived_class is derived from the Base_class. The visibility_mode could be public, private or protected. The default visibility mode (if not stated) is private.

Example:

```
class Person
{
   // data members and member functions
};
class Student : public  Person
{
  // data members and member functions
};
```

Visibility modes in inheritance: These modes specify how the features of the base class are derived to the base class. It controls the accessibility of the private, protected and public members of the base class while deriving on the base class. The visibility modes could be private, protected or public.

a) Public: If the visibility mode is public, the members in the derived class remain as it is as that of base class. That means, public members of the base class remain public in the derived class and the protected members of the base class remain protected in the derived class.

By: Manoj Sapkota, U-TEC

b) Protected: If the visibility mode is protected, the public and the protected members of the base class become protected in the derived class.
c) Private: If the visibility mode is private, the protected and the public members of the base class become private in the derived class.

Note: Private members of the base class are not accessible to the derived class.

Summary Table:

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

Example:

```
class NPI {
  private:
    int a;
  protected:
    int b;
  public:
    int c;
};
class Utec : private NPI {
  //a is not accessible
  //b is private
  //c is private
};
class Kabilas_resort : protected NPI {
  //a is not accessible
  //b is protected
  //c is protected
};
class NS_hospital : public NPI {
  //a is not accessible
  //b is protected
  //c is public
};
```

By: Manoj Sapkota

Types of inheritance: According to the level and nature of function or data sharing, inheritance is classified into following types:

1. Single inheritance
2. Multiple inheritance
3. Multi-level inheritance
4. Hierarchical inheritance
5. Hybrid inheritance

1. Single inheritance: It is the type of inheritance in which one derived class is inherited from the only one base class.

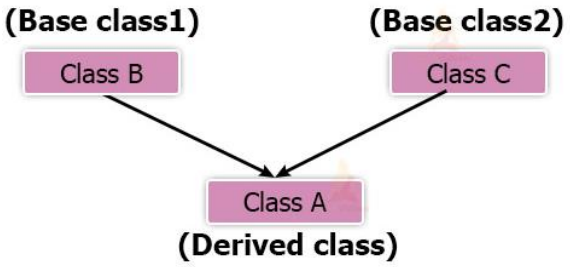
- All the other types of inheritances are based on single inheritance.

General form:

```
class Base_class
{
   // body of Base_class
};
class Derived_class : visibility_mode  Base_class
{
  // body of Derived_class
};
```

An example of a single inheritance is shown below. *[Example program- 27]*

2. Multiple inheritance: It is the type of inheritance in which one derived class inherits its features from two or more than two base classes. The derived class will share the features from the base classes as per its need.



General form:

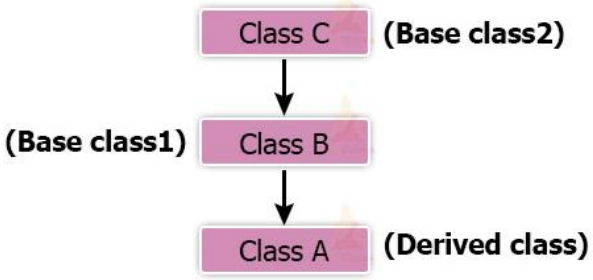
```
class Base_class1
{
   // body of Base_class1
};
```

By: Manoj Sapkota, U-TEC

```
class Base_class2
{
   // body of Base_class2
};
class Derived_class : visibility_mode  Base_class1, visibility_mode Base_class2
{
  // body of Derived_class
};
```

An example of a multiple inheritance is shown below. *[Example program- 28]*

3. Multi-level inheritance: When one class inherits another class which is further inherited by another by another class, it is called multi-level inheritance. That means, in this type of inheritance, one derived class inherits from another derived class.

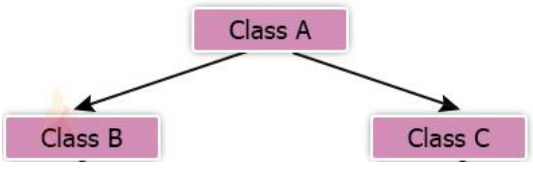


General form:

```
class Base_class2
{
   // body of Base_class2
};
class Base_class1 : visibility_mode Base_class2
{
   // body of Base_class1
};
class Derived_class : visibility_mode  Base_class1
{
  // body of Derived_class
};
```

An example of a multi-level inheritance is shown below. *[Example program- 29]*

4. Hierarchical inheritance: When more than one class is inherited from a single base class, it is called hierarchical inheritance. The derived classes will share the features from the base class as per their need.

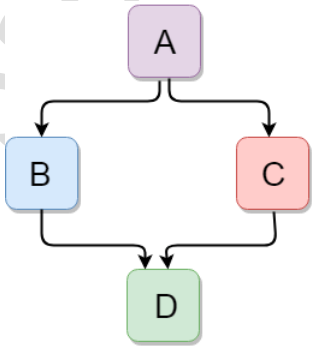


General form:

```
class A
{
   // body of class A
};
class B : visibility_mode A
{
   // body of class B
};
class C : visibility_mode A
{
  // body of class C
};
```

An example of a hierarchical inheritance is shown below. *[Example program- 30]*

5. Hybrid inheritance: When two or more than two types of inheritances are combined, it is known as hybrid inheritance. It could be the combination of multiple and multi-level, multiple and hierarchical, multi-level and hierarchical or any other combinations of inheritance. An example of a hybrid inheritance is shown in figure, which is a combination of hierarchical & multiple inheritance.



General form:

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

An example of a hybrid inheritance is shown below. *[Example program- 31]*

Example program- 27: (Single Inheritance) [try yourself]
Example program- 28: (Multiple inheritance) [try yourself]
Example program- 29: (Multi-level inheritance) [try yourself]
Example program- 30: (Hierarchical inheritance) [try yourself]
Example program- 31: (Hybrid inheritance) [try yourself]

Function overriding: Functions overriding is a feature that allows us to use a function in the derived class that is already present in its base class. If a derived class has a member function with the same name and signature as in the base class, then it is known as function overriding. This feature allows us to override any functionality of the base class in the derived class.

Function overriding allows us to create a newer version of the base class function in the derived class. If we call the overridden function using the object of the derived class, the function of the derived class is executed.

An example of a use of function overriding is shown below. *[Example program-32]*

Example program- 32:

```
class A{
   public:
     void fun(){
       cout<<"Fun from class A."<<endl;
     }
};

class B : public A{
   public:
     void fun(){ //function overriding
       cout<<"Fun from class B."<<endl;
     }
};

int main()
{
  B obj;
  obj.fun(); //calls fun() from class B due to function overriding
  return 0;
}
```

Output: Fun from class B.

Explanation: In the above program, the base class A and the derived class B have the same function fun( ) with same name and signature. Now, when this function is called from the derived object, the fun( ) in the derived class is executed as it overrides the same in base class.

Ambiguity in multiple inheritance: One of the common errors that might occur during multiple inheritance is the ambiguity error. It arises during function overriding in multiple inheritance. For example, two base classes have a same function which is not overridden in derived class. And if we try to call the function using the object of the derived class, the compiler won't be able to know which function to call and shows error due to ambiguity.

An example that shows ambiguity in multiple inheritance is given below. *[Example program- 33]*

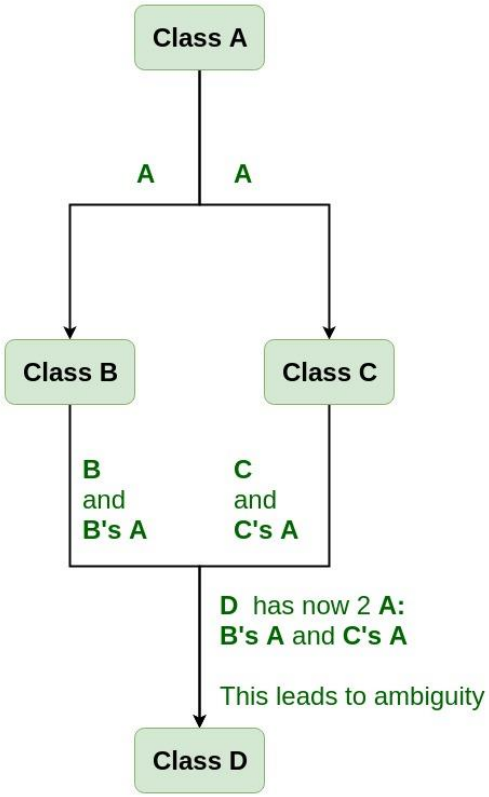
```
class Russia{
  public:
    void acceptWar(){
     cout<<"Hey! you can't join alliance with NATO."<<endl;
    }
};
class Ukraine{
  public:
   void acceptWar(){
    cout<<"Sorry! I don't like you. I prefer to join them."<<endl;
   }
};
class World: public Russia, public Ukraine{
  public:
    void stopWar(){
     cout<<"Hey please! Stop war."<<endl;
    }
};
int main(){
  World w;
  w.acceptWar();     //Error due to ambiguity
  return 0;
}
```

Explanation: In the above program, the two base classes 'Russia' and 'Ukraine' have same function acceptWar( ), and is absent (is not overridden) in derived class 'World'. If this function is called by creating an object of derived class 'World', then the compiler gets confused which function of the base class to call. This is the ambiguity problem.

Solution to the ambiguity problem:
The problem can be solved by using the scope resolution operator to specify which function of the base class is being called. In the above example, ambiguity problem can be solved by writing the following code.

```
int main(){
  World w;
  w.Russia::acceptWar();   //solves the problem of ambiguity
  w.Ukraine::acceptWar();  //solves the problem of ambiguity
  return 0;
}
```

Ambiguity in hybrid inheritance: Let us consider a following situation in hybrid inheritance.



Here, both the class B and class C are inheriting the features from class A. They both have the single copy of class A. Both these classes are inherited further into another class D. So, the features from class A are inherited twice to class D. The class D will have two copies of A, one from class B and another from class C. Now, if any member of class A is accessed by an object of class D, the compiler gets confused which path should it follow either from B, or from C and hence displays error. This is the ambiguity problem resulting in hybrid inheritance.

An example of hybrid inheritance that results into ambiguity error is given below. *[Example program- 34]*

Example program- 34:

```
class A{
   public:
     void fun(){
       cout<<"Fun from class A."<<endl;
     }
};
class B : public A{

};
class C : public A{

};
class D: public B, public C{

};
int main()
{
  D obj;
  obj.fun(); /** It results ambiguity error **/
  return 0;
}
```

Solution to the ambiguity problem:
The problem can be solved by using a **virtual base class**. Virtual base class prevents multiple copies of a given class appearing in an inheritance hierarchy when using multiple inheritances. That means, only one copy of the base class is inherited to its derived classes.

This is obtained by declaring the base class as a virtual class during inheritance. It is specified by placing a 'virtual' keyword in the base class. In the above example, a base class could be declared as virtual as shown below.

```
class A{

};
class B: virtual public A {

};
class C: virtual public A {

};
class D: public B, public C {

};
```

By: Manoj Sapkota, U-TEC

A use of virtual base class that solves the ambiguity problem occurred in program-34 is shown below.

```
class A{
   public:
     void fun(){
       cout<<"Fun from class A."<<endl;
     }
};
//Making base class virtual solves the ambiguity problem
class B : virtual public A{

};
class C : virtual public A{

};
class D: public B, public C{

};
int main()
{
  D obj;
  obj.fun();
  return 0;
}
```

Output: Fun from class A.

Constructors in inheritance: When an object of a derived class is created, the default constructor of the derived class will be invoked. But before the execution of the constructor of the derived class, the default constructor of all the base classes will be invoked and executed implicitly (automatically). Therefore in this process, the order of execution is the constructor of the base class at first and then the derived class.

If the base class has parameterized constructors, then they should be explicitly called from the constructor of the derived class using the following syntax:

Derived_class_constructor : Base_class_constructor(arguments)

An example that demonstrates the order of execution of constructors in base and derived class is as follows: *[Example program- 35]*

Example program- 35:

```
class Russia{
  public:
    Russia(){
     cout<<"I am constructor of Russia."<<endl;
    }
};
class Ukraine: public Russia{
  public:
   Ukraine(){
    cout<<"I am constructor of Ukraine"<<endl;
   }
};
int main(){
  Ukraine u;
  return 0;
}
```

Output: I am constructor of Russia.
I am constructor of Ukraine.

Explanation: When an object of derived class Ukraine is created, both the constructors of the base class and the derived class are executed automatically. At first, the constructor of the base class Russia is executed and then the constructor of the derived class Ukraine. An example that demonstrates the case of base class having parameterized constructor is shown below. *[Example program- 36]*

Example program- 36:

```
class Russia{
  int a;
  public:
  Russia(int y){
   a=y;
   cout<<"Russian constructor, Value: "<<a<<endl;
  }
};
class Ukraine: public Russia{
  int b;
  public:
  Ukraine(int x, int y): Russia(y){
   b=x;
   cout<<"Ukrainian constructor, Value: "<<b<<endl;
  }
};
int main(){
  Ukraine u(20,70);
  return 0;
}
```

Output: Russian constructor, Value: 70
Ukrainian constructor, Value: 20

Sapkota, U-TEC