

Subject :- Object Oriented Programming

Unit 2 :-

Inheritance and Pointers

The capability of a class to derive properties and characteristics from another class is called **Inheritance**.

Inheritance is one of the most important feature of Object Oriented Programming.

## Subject :- Object Oriented Programming

### Unit 2

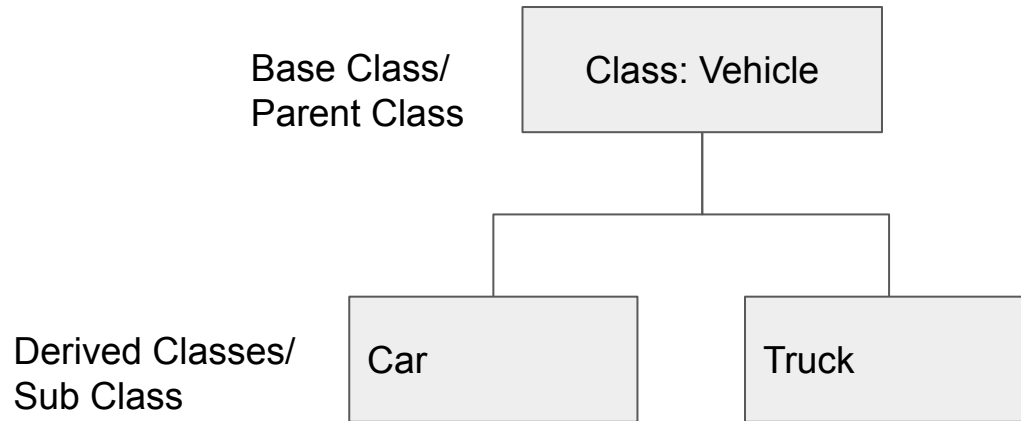
Class: Vehicle

Select any vehicle in your mind?

What's the difference between the vehicles

## Subject :- Object Oriented Programming

### Unit 2



## Subject :- Object Oriented Programming

### Unit 2

**Sub Class:** The class that inherits properties from another class is called Subclass or Derived Class.

**Super Class:** The class whose properties are inherited by subclass is called Base Class or Super class.

## Subject :- Object Oriented Programming

### Unit 2

#### Why and when to use inheritance?

Consider a group of vehicles. You need to create classes for Bus, Car and Truck. The methods fuelAmount(), capacity(), applyBrakes() will be same for all of the three classes. If we create these classes avoiding inheritance then we have to write all of these functions in each of the three classes as shown in below figure:

#### Class Bus

```
fuelAmount()  
capacity()  
applyBrakes()
```

#### Class Car

```
fuelAmount()  
capacity()  
applyBrakes()
```

#### Class Truck

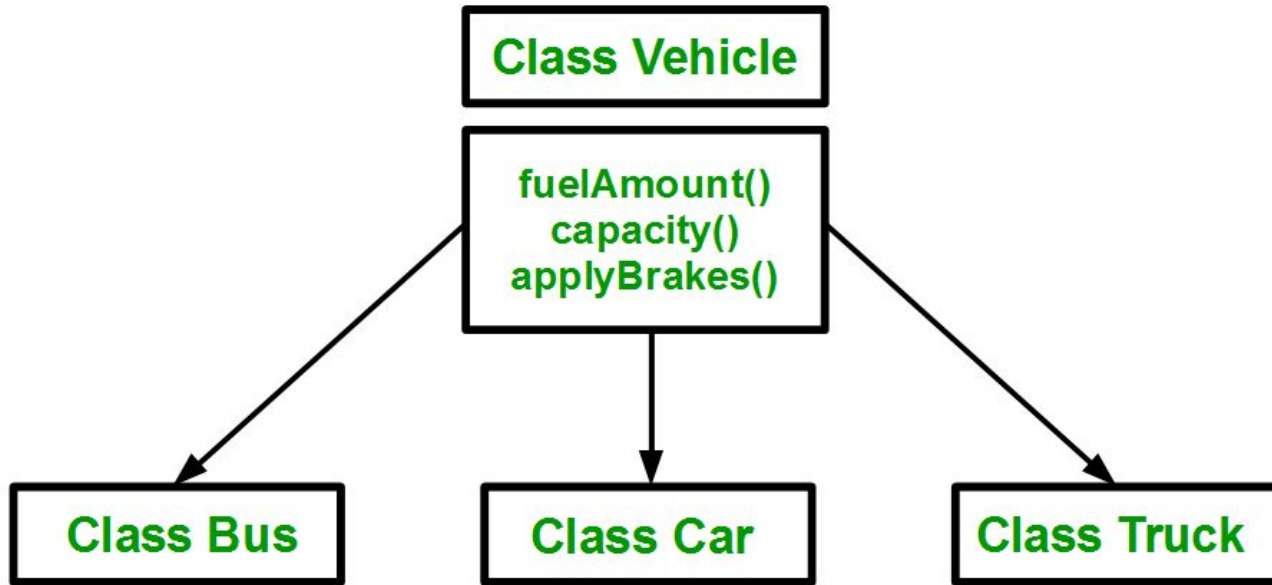
```
fuelAmount()  
capacity()  
applyBrakes()
```

**is-a relationship** - Inheritance is an is-a relationship. We use inheritance only if an is-a relationship is present between the two classes.

## Subject :- Object Oriented Programming

### Unit 2

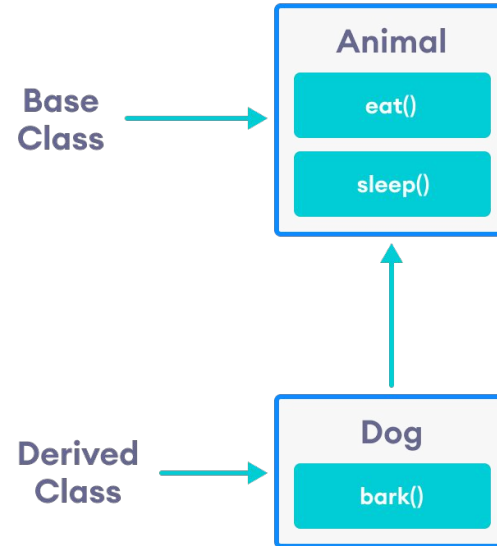
Why and when to use inheritance?



## Subject :- Object Oriented Programming

### Unit 2

#### Inheritance Example



## Subject :- Object Oriented Programming

### Unit 2

#### **Inheritance Example**

Here are some examples:

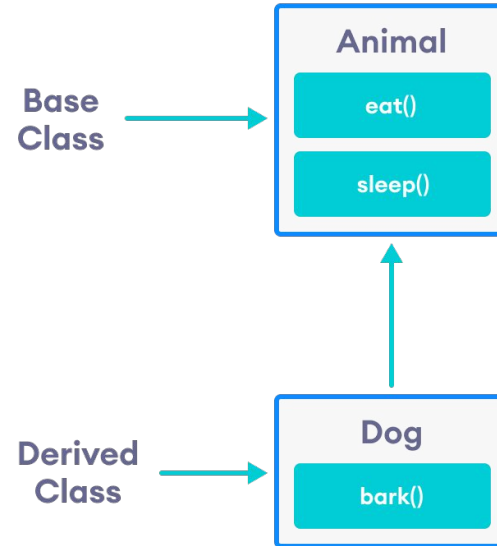
- A car is a vehicle.
- Orange is a fruit.
- A surgeon is a doctor.
- A dog is an animal.



## Subject :- Object Oriented Programming

### Unit 2

Lets try to implement this example



## **Subject :- Object Oriented Programming**

### **Unit 2**

#### **Access Specifiers in Inheritance:-**

- 1. Private**
- 2. Protected**
- 3. Public**

## Subject :- Object Oriented Programming

### Unit 2

#### Access Control and Inheritance

A derived class can access all the non-private members of its base class. Thus base-class members that should not be accessible to the member functions of derived classes should be declared private in the base class.

We can summarize the different access types according to - who can access them in the following way

—

Access	public	protected	private
Same class	yes	yes	yes
Derived classes	yes	yes	no
Outside classes	yes	no	no

# The private members of the base class are never inherited.

- When the base class is privately inherited by the derived class, public members of the base class becomes the private members of the derived class. Therefore, the public members of the base class are not accessible by the objects of the derived class only by the member functions of the derived class.
- When the base class is publicly inherited by the derived class, public members of the base class also become the public members of the derived class. Therefore, the public members of the base class are accessible by the objects of the derived class as well as by the member functions of the base class.

# How to make a Private Member Inheritable

- The private member is not inheritable. If we modify the visibility mode by making it public, but this takes away the advantage of data hiding.
- C++ introduces a third visibility modifier, i.e., **protected**. The member which is declared as protected will be accessible to all the member functions within the class as well as the class immediately derived from it.
- **Public**: When the member is declared as public, it is accessible to all the functions of the program.
- **Private**: When the member is declared as private, it is accessible within the class only.
- **Protected**: When the member is declared as protected, it is accessible within its own class as well as the class immediately derived from it.

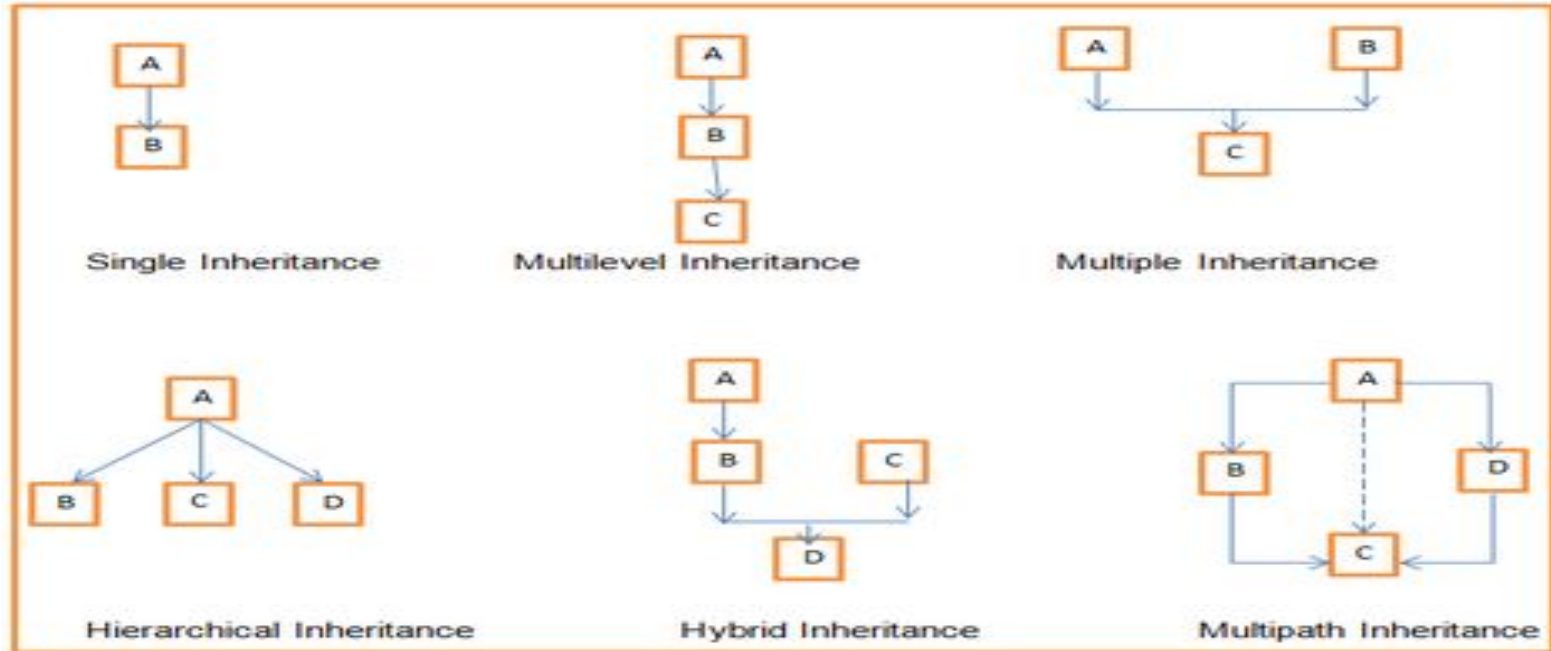
# Syntax:

- Class parent
- {
- body
- };
- Class child : public parent
- {
- Body
- };

## Subject :- Object Oriented Programming

### Unit 2

#### Types of Inheritance



## Subject :- Object Oriented Programming

### Unit 2 :-

### Constructors and Destructors in Inheritance

**Base class constructors are always called in the derived class constructors.**

Whenever you create derived class object, first the base class default constructor is executed and then the derived class's constructor finishes execution.

Note:-

Whether derived class's default constructor is called or parameterised is called, base class's default constructor is always called inside them.



**Subject :- Object Oriented Programming**

**Unit 2 :-**

**Constructors and Destructors in Inheritance**

**Case 1:-**

Base class Default Constructor in Derived class Constructors

**Case 2:-**

Base class Parameterized Constructor in Derived class Parameterized Constructors

## Subject :- Object Oriented Programming

### Unit 2 :-

### Constructors and Destructors in Inheritance

```
#include<iostream>
using namespace std;
class parent
{

    int x;
public:
    // parameterized constructor
    parent(int i)
    {
        x = i;
        cout << "Parent class
Parameterized Constructor\n";
    }
};
```

```
class child: public parent
{
    int y;
public:

    // parameterized constructor
    child(int j) : parent(j)
//Explicitly calling
    {
        y = j;
        cout << "Child class
Parameterized Constructor\n";
    }
};

int main()
{
    child c(10);    return 0;
}
```

Subject :- Object Oriented Programming

Unit 2 :-

Constructors and Destructors in Inheritance

**Destructors:-**

**Destructors** in C++ are called in the opposite order of that of Constructors.

## Subject :- Object Oriented Programming

### Unit 2 :-

### Constructors and Destructors in Inheritance

#### Order of Inheritance



#### 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)

## Subject :- Object Oriented Programming

### Unit 2 :-

#### ***Overriding member functions using Inheritance***

1. **Inheritance** is a feature of OOP that allows us to create derived classes from a base class. The derived classes inherit features of the base class.
2. Suppose, the same function is defined in both the derived class and the based class. Now if we call this function using the object of the derived class, the function of the derived class is executed.
3. This is known as function overriding in C++. The function in derived class overrides the function in base class.

## Subject :- Object Oriented Programming

### Unit 2 :-

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

class Base {
public:
    void print() {
        cout << "Base Function" <<
endl;
    }
};

class Derived : public Base {
public:
    void print() {
        cout << "Derived Function"
<< endl;
    }
};
```

```
int main() {
    Derived derived1;
    derived1.print();
    return 0;
}
```

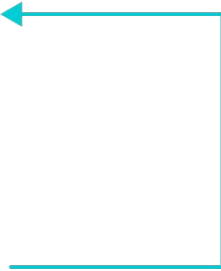
## Subject :- Object Oriented Programming

### Unit 2 :-

```
class Base {
    public:
        void print() {
            // code
        }
};

class Derived : public Base {
    public:
        void print() {
            // code
        }
};

int main() {
    Derived derived1;
    derived1.print();
    return 0;
}
```



Here, the same function `print()` is defined in both Base and Derived classes.

So, when we call `print()` from the Derived object `derived1`, the `print()` from Derived is executed by overriding the function in Base.

Subject :- Object Oriented Programming

Unit 2 :-

## Access Overridden Function in C++

To access the overridden function of the base class, we use the scope resolution operator `::`.

Ex. `derived2.Base::print();`



# Subject :- Object Oriented Programming

## Unit 2 :-

```
class Base {  
    public:  
    void print() {  
        // code  
    }  
};  
  
class Derived : public Base {  
    public:  
    void print() {  
        // code  
    }  
};  
  
int main() {  
    Derived derived1, derived2;  
  
    derived1.print();  
  
    derived2.Base::print();  
  
    return 0;  
}
```

The diagram illustrates the resolution of the `derived2.Base::print();` statement. A blue line originates from the `print()` part of the statement and points to the `print()` method definition within the `Base` class, indicating that this specific call bypasses the `Derived` class's `print()` method and directly invokes the base class's implementation.

Here, this statement

```
derived2.Base::print();
```

accesses the `print()` function of the Base class.

## Subject :- Object Oriented Programming

### Unit 2 :-

#### C++ Call Overridden Function From Derived Class

```
/ C++ program to call the  
overridden function  
// from a member function of  
the derived class
```

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

```
class Base {  
    public:  
        void print() {  
            cout << "Base Function"  
<< endl;  
        }  
};
```

```
class Derived : public Base {  
    public:  
        void print() {  
            cout << "Derived Function"  
<< endl;
```

```
        // call overridden function  
        Base::print();  
    }  
};
```

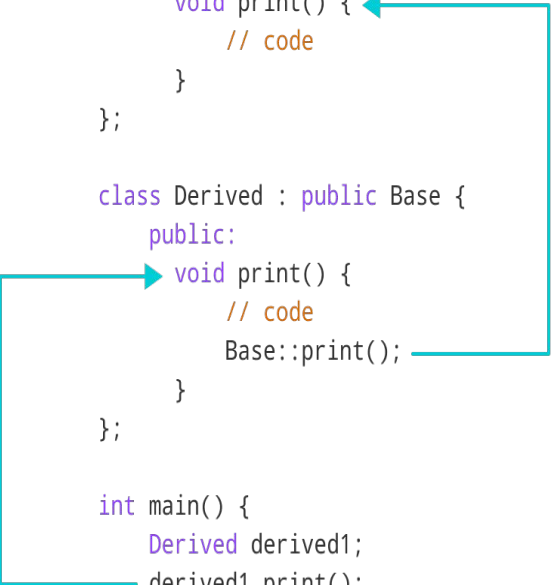
```
int main() {  
    Derived derived1;  
    derived1.print();  
    return 0;  
}
```

# Subject :- Object Oriented Programming

## Unit 2 :-

### C++ Call Overridden Function From Derived Class

```
class Base {  
    public:  
    void print() {  
        // code  
    }  
};  
  
class Derived : public Base {  
    public:  
    void print() {  
        // code  
        Base::print();  
    }  
};  
  
int main() {  
    Derived derived1;  
    derived1.print();  
    return 0;  
}
```



The diagram consists of two blue arrows. One arrow starts from the `Base::print();` line inside the `Derived::print()` function and points to the `void print() {` line in the `Base` class definition. The second arrow starts from the `derived1.print();` line in the `main()` function and points to the `void print() {` line in the `Derived` class definition.

In this program, we have called the overridden function inside the Derived class itself.

```
class Derived : public Base {  
    public:  
    void print() {  
        cout << "Derived Function" << endl;  
        Base::print();  
    }  
};
```

Notice the code `Base::print();`, which calls the overridden function inside the Derived class.

# Subject :- Object Oriented Programming

## Unit 2 :-

### Public and Private Inheritance

In C++ inheritance, we can derive a child class from the base class in different access modes.

For example,

```
class Base {  
    . . . . .  
};  
  
class Derived : public Base {  
    . . . . .  
};
```

Notice the keyword `public` in the code

```
class Derived : public  
Base
```

Notice the keyword `public` in the code `class Derived : public Base`

## Subject :- Object Oriented Programming

### Unit 2 :-

#### Public and Private Inheritance

This means that we have created a derived class from the base class in **public mode**. Alternatively, we can also derive classes in **protected or private** modes.

These 3 keywords (`public`, `protected`, and `private`) are known as access specifiers in C++ inheritance.

## Subject :- Object Oriented Programming

### Unit 2 :-

## public, protected and private inheritance in C++

public, protected, and private inheritance have the following features:

- public inheritance makes public members of the base class public in the derived class, and the protected members of the base class remain protected in the derived class.
- protected inheritance makes the public and protected members of the base class protected in the derived class.
- private inheritance makes the public and protected members of the base class private in the derived class
- Note: private members of the base class are inaccessible to the derived class..

## Subject :- Object Oriented Programming

### Unit 2 :-

Base class member access specifier	Type of Inheritance		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)

## Subject :- Object Oriented Programming

### Unit 2 :-

#### Access control (or) visibility mode

contd..

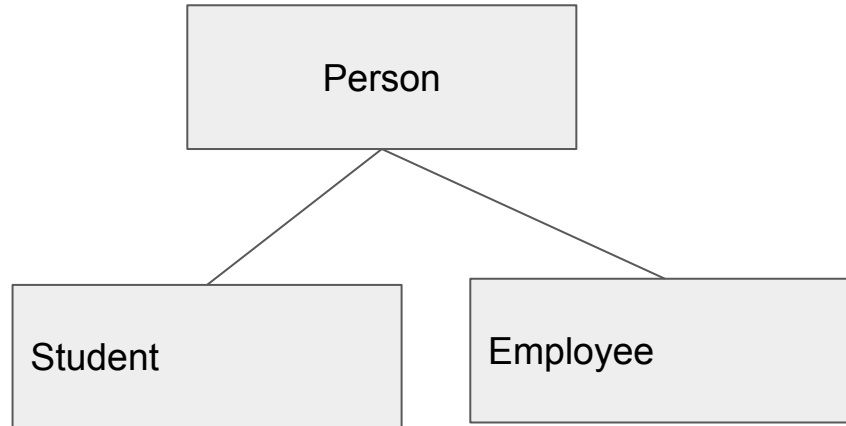
Inheritance type	case class	derived class
private	private numbers public numbers Protected numbers	not inherited private numbers private numbers
public	private numbers public numbers protected numbers	not inherited public numbers protected numbers
protected	private numbers public numbers protected numbers	not inherited protected numbers protected numbers



Subject :- OOPCG Lab

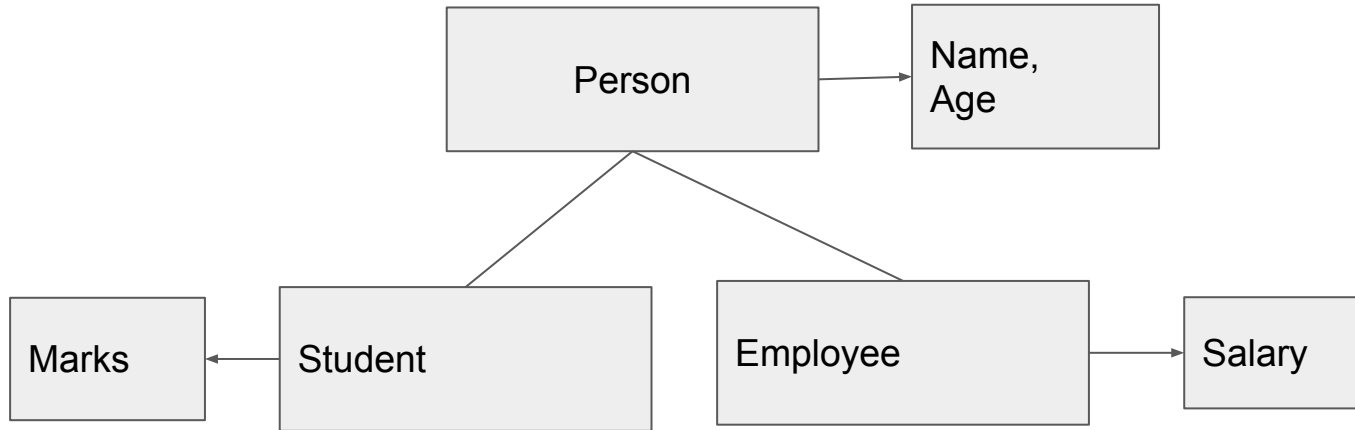
Assignment 1

## Example of Inheritance



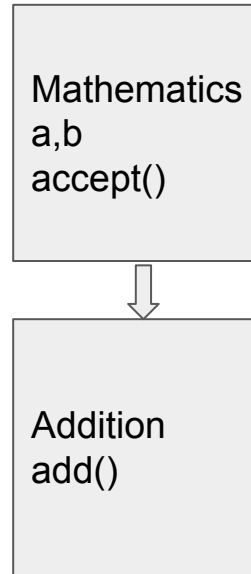
## Subject :- OOPCG Lab

### Hierarchical Inheritance



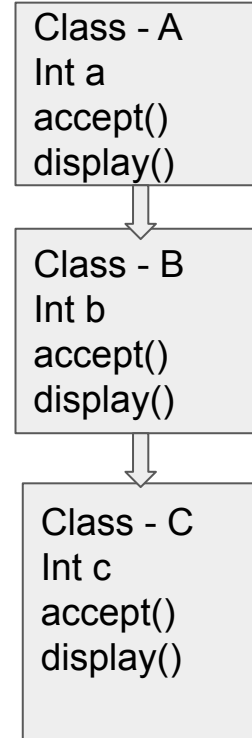
## Subject :- OOPCG Lab

### Single Inheritance



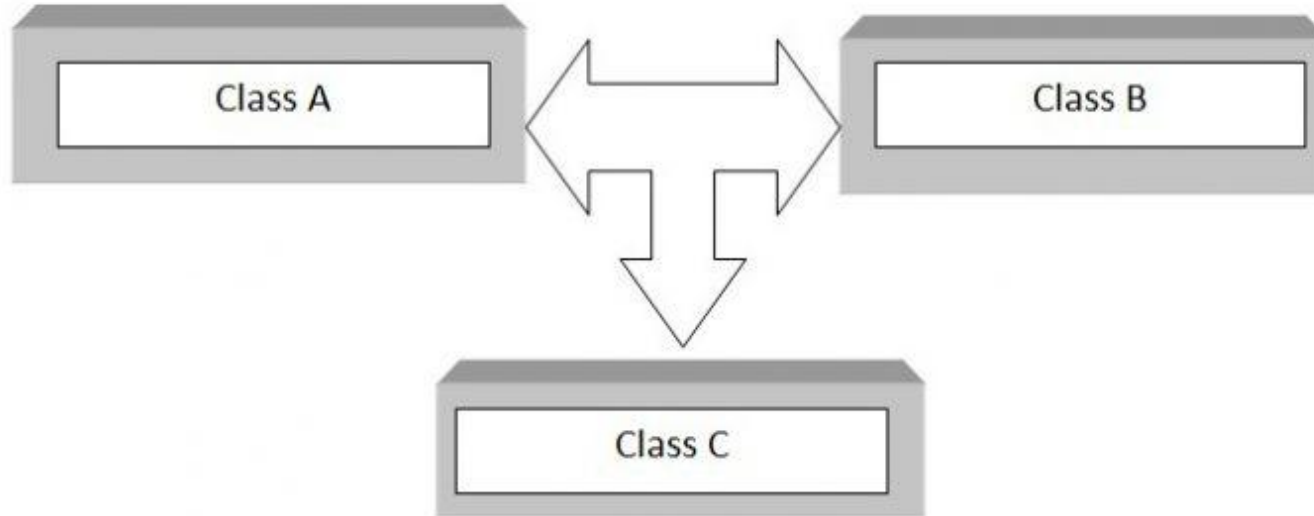
**Subject :- OOPCG Lab**

**Multilevel Inheritance**



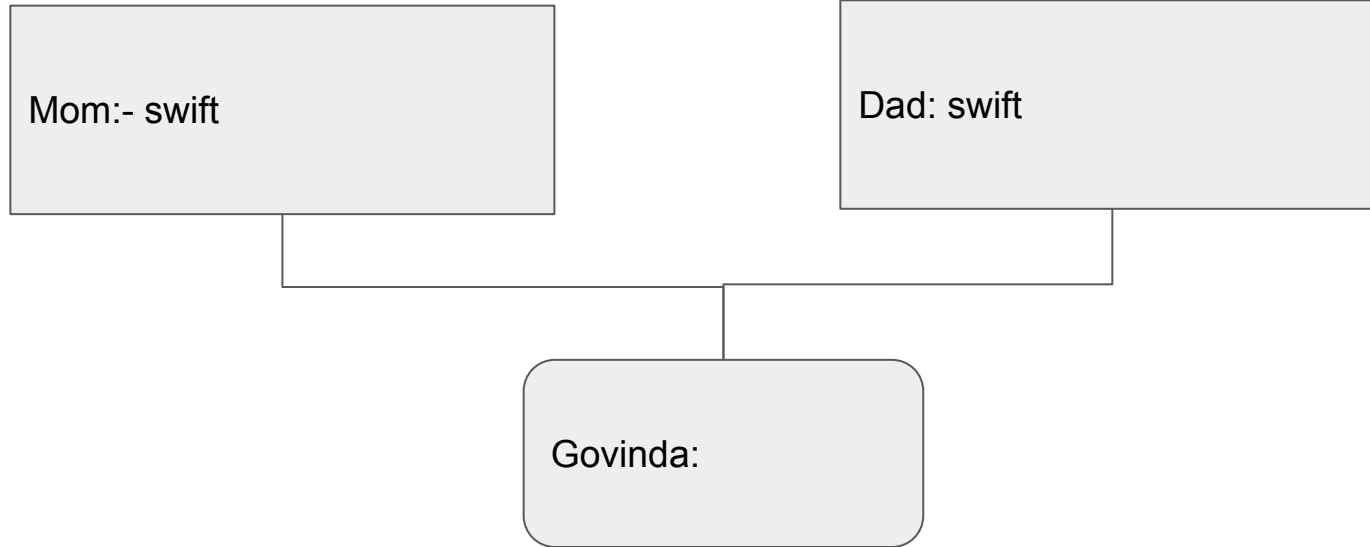
## Subject :- OOPCG Lab

### Multiple Inheritance



## Subject :- OOPCG Lab

### Multiple Inheritance



## Subject :- OOPCG Lab

### Ambiguity in Multiple Inheritance

The most obvious problem with multiple inheritance occurs during function overriding.

Suppose, two base classes have a same function which is not overridden in derived class.

If we try to call the function using the object of the derived class, compiler shows error. It's because compiler doesn't know which function to call.

```
class base1
{
    public:
        void someFunction( )
        { .... .. }
};
class base2
{
    void someFunction( )
    { .... .. }
};
```

```
class derived : public base1, public base2
{
};

int main()
{
    derived obj;

    obj.someFunction() // Error!
}
```

## Subject :- OOPCG Lab

### Ambiguity in Multiple Inheritance

This problem can be solved using scope resolution function to specify which function to class either

```
base1or base2
```

```
int main()
```

```
{
```

```
    obj.base1::someFunction( ); // Function of base1 class is called
```

```
    obj.base2::someFunction(); // Function of base2 class is called.
```

```
}
```

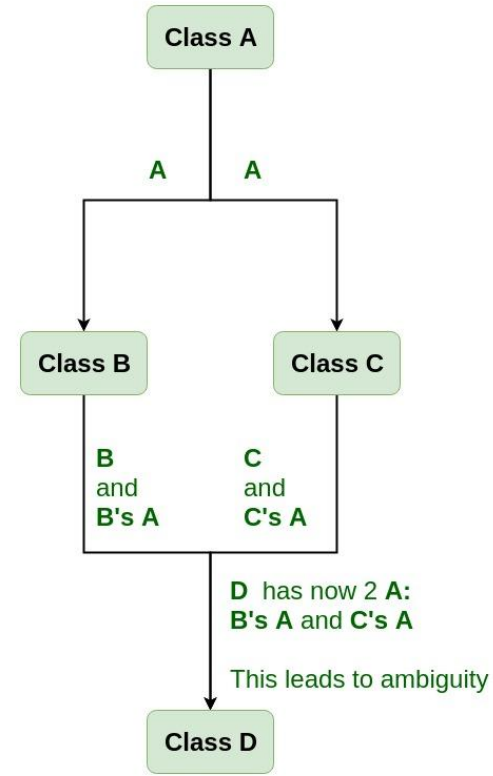


## Subject :- OOPCG Lab

### Virtual Base Class

Virtual base classes are used in virtual inheritance in a way of preventing multiple “instances” of a given class appearing in an inheritance hierarchy when using multiple inheritances. **Need for Virtual Base Classes:**

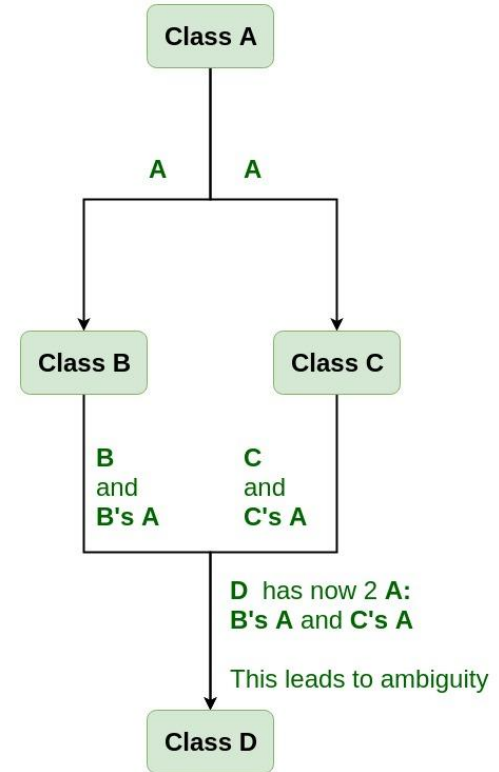
Consider the situation where we have one class **A**. This class is **A** is inherited by two other classes **B** and **C**. Both these class are inherited into another in a new class **D** as shown in figure below.



## Subject :- OOPCG Lab

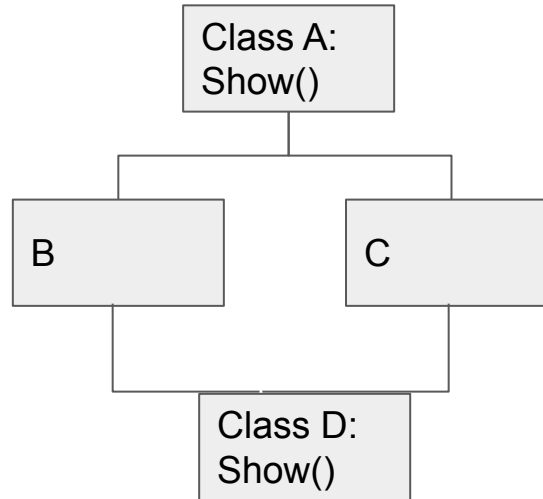
### Virtual Base Class

As we can see from the figure that data members/function of class **A** are inherited twice to class **D**. One through class **B** and second through class **C**. When any data / function member of class **A** is accessed by an object of class **D**, ambiguity arises as to which data/function member would be called? One inherited through **B** or the other inherited through **C**. This confuses compiler and it displays error.



## Subject :- OOPCG Lab

### Virtual Base Class



**It Creates an Ambiguity when we call show method from object of D class as it has 2 paths from Base class A ie. A-B-D and A-C-D**

## Subject :- OOPCG Lab

### Virtual Base Class

#### How to resolve this issue?

To resolve this ambiguity when class **A** is inherited in both class **B** and class **C**, it is declared as **virtual base class** by placing a keyword **virtual** as :

Syntax for Virtual Base Classes:

#### Syntax 1:

```
class B : virtual public A
{
};
```

#### Syntax 2:

```
class C : public virtual A
{
};
```

## Subject :- OOP

### Friend Class

As we know that a class cannot access the private members of other class.

Similarly a class that doesn't inherit another class cannot access its protected members.

Data hiding is a fundamental concept of object-oriented programming. It restricts the access of private members from outside of the class.

#### Friend Class:

A **friend class** is a class that can access the **private** and **protected** members of a class in which it is declared as **friend**. This is needed when we want to allow a **particular class to access the private and protected members of a class**.

## Subject :- OOP

### Friend Class

We can also use a friend Class in C++ using the `friend` keyword. For example,

```
Class A
{
    Friend class B;
}
Class B
{
    //here class B can access private data members of class A
}
```

Since `classB` is a friend class, we can access all members of `classA` from inside `classB`.

## Subject :- OOP

### Friend Class

```
#include <iostream>
using namespace std;
class XYZ {
private:
    char ch='A';
    int num = 11;
public:
    /* This statement would make class ABC
    * a friend class of XYZ, this means that
    * ABC can access the private and protected
    * members of XYZ class.
    */
    friend class ABC;
};
class ABC {
public:
    void disp(XYZ obj){
        cout<<obj.ch<<endl;
        cout<<obj.num<<endl;
    }
};
```

```
int main() {
    ABC obj;
    XYZ obj2;
    obj.disp(obj2);
    return 0;
}
```

## Subject :- OOP

### Pointers

What are Pointers?

A pointer is a variable whose **value is the address** of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before using it to store any variable address. The general form of a pointer variable declaration is –

```
datatype *pointer_name;
```

```
int *ip    // pointer to integer variable
```

```
float *fp;  // pointer to float variable
```

```
double *dp; // pointer to double variable
```

```
char *cp;   // pointer to char variable
```



## Subject :- OOP

### Pointers

## Initialization of Pointer variable

**Pointer Initialization** is the process of assigning address of a variable to a **pointer** variable. Pointer variable can only contain address of a variable of the same data type. In C++ language **address operator &** is used to determine the address of a variable. The **&** (immediately preceding a variable name) returns the address of the variable associated with it.

```
void main()
{
    int a = 10;
    int *ptr;    //pointer declaration
    ptr = &a;    //pointer initialization
}
```

## Subject :- OOP

### Pointers

#### Pointers to Objects

```
class Simple
{
    public:
    int a;
};

int main()
{
    Simple obj;
    Simple* ptr; // Pointer of class type
    ptr = &obj;
```

```
    cout << obj.a;
    cout << ptr->a; // Accessing member
    with pointer
}
```

## Subject :- OOP

### Pointers

#### This Pointer

- C++ provides a keyword 'this', which represents the current object and passed as a hidden argument to all member functions.
- The **this** pointer is a constant pointer that holds the memory address of the **current object**.
- The **this** pointer is not available in static member functions as static member functions can be called without any object. static member functions can be called with class name.

## Subject :- OOP

### Pointers

#### This Pointer

```
#include <iostream>
#include <conio.h>
using namespace std;

class sample
{
    int a,b;
    public:
        void input(int a,int b)
        {
            a=this->a;
            b=this->b;
        }
        void output()
        {
            cout<<"a = "<<a<<endl<<"b = "<<b;
        }
};
```

```
int main()
{
    sample x;
    x.input(5,8);
    x.output();
    getch();
    return 0;
}
```

**Subject :- OOP**

## **Pointers**

This Pointer

### **Assignment**

- Write a Program in CPP To Create a class Employee having fields emp\_id,emp\_name,emp\_salary accept this data using parameterized constructor and display it. Make use of This pointer.

## Subject :- OOP

### Pointers

#### Pointers vs Arrays

In C++, **Pointers** are variables that hold addresses of other variables. Not only can a pointer store the address of a single variable, it can also store the address of cells of an **array**.

Consider this example:

```
int *ptr;  
int arr[5];  
  
// store the address of the first  
// element of arr in ptr  
ptr = arr;
```

Here, `ptr` is a pointer variable while `arr` is an `int` array. The code `ptr = arr;` stores the address of the first element of the array in variable `ptr`.

## Subject :- OOP

### Pointers

#### Pointers vs Arrays

```
int *ptr;  
int arr[5];  
ptr = &arr[0];
```

Notice that we have used `arr` instead of `&arr[0]`. This is because both are the same. So, the code below is the same as the code above.

The addresses for the rest of the array elements are given by `&arr[1]`, `&arr[2]`, `&arr[3]`, and `&arr[4]`.

## Subject :- OOP

### Pointers vs Arrays

## Point to Every Array Elements

Suppose we need to point to the fourth element of the array using the same pointer `ptr`.

Here, if `ptr` points to the first element in the above example then `ptr + 3` will point to the fourth element. For example,

```
int *ptr;  
int arr[5];  
ptr = arr;
```

```
ptr + 1 is equivalent to &arr[1];  
ptr + 2 is equivalent to &arr[2];  
ptr + 3 is equivalent to &arr[3];  
ptr + 4 is equivalent to &arr[4];
```



## Subject :- OOP

### Pointers vs Arrays

#### Point to Every Array Elements

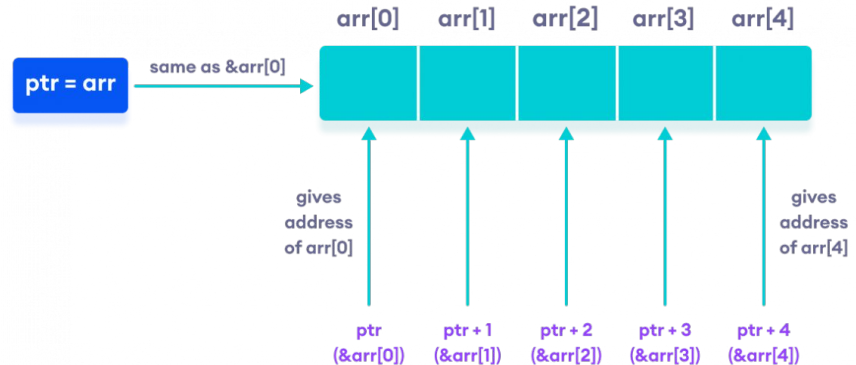
Suppose if we have initialized `ptr = &arr[2];` then

`ptr - 2` is equivalent to `&arr[0];`

`ptr - 1` is equivalent to `&arr[1];`

`ptr + 1` is equivalent to `&arr[3];`

`ptr + 2` is equivalent to `&arr[4];`



## Subject :- OOP

# Pointers to Functions

As we know that pointers are used to point some variables;

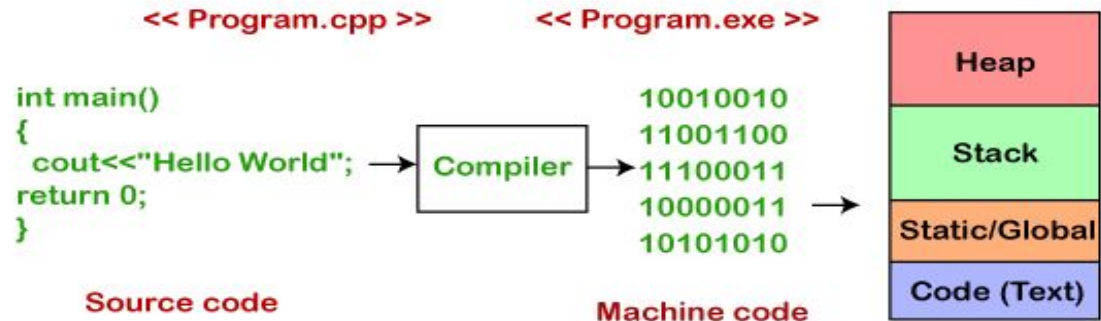
similarly, the **function pointer is a pointer used to point functions**. It is basically used to store the address of a function.

We can call the function by using the function pointer,

## Function Pointers

What would be the address of a function?

What is the address of a function?



Subject :- OOP

## Pointers to Functions

### Syntax for Declaration

The following is the syntax for the declaration of a function pointer:

Return-type(\*function Pointer)(parameters if any);

Ex.

```
int (*FuncPtr) (int,int);
```

## Subject :- OOP

### Pointers to Function

```
#include <iostream>

using namespace std;

int add(int a , int b)
{
    return a+b;
}
```

```
int main()
{
    int (*funcptr)(int,int); // function pointer
    declaration
    funcptr=&add; // funcptr is pointing to the
    add function
    int sum=funcptr(5,5);
    cout << "value of sum is :" <<sum;
    return 0;
}
```

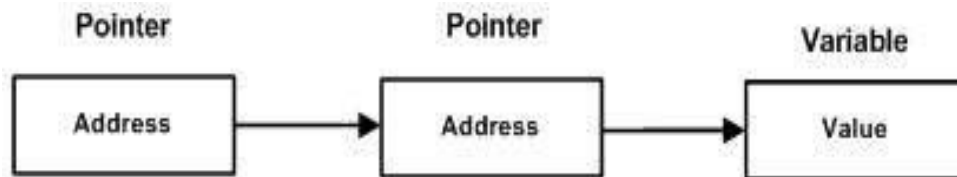
## Subject :- OOP

### Pointers to Pointers

A pointer to a pointer is a form of multiple indirection or a chain of pointers.

Normally, a pointer contains the address of a variable.

When we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value as shown below.



## Subject :- OOP

### Pointers to Pointers

A variable that is a pointer to a pointer must be declared as such. This is done by placing an additional asterisk in front of its name. For example, following is the declaration to declare a pointer to a pointer of type int –

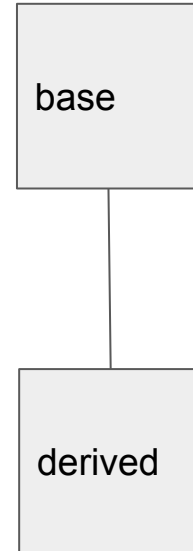
```
int **var;
```

## Subject :- OOP

### Pointers to Derived Class

In C++, we can declare a pointer points to the base class as well as derived class.

```
base b;  
base *bptr;  
bptr=&b;  
derive d;  
bptr=&d;
```



## Subject :- OOP

### Null Pointer

- It is always a good practice to assign the pointer NULL to a pointer variable in case you do not have exact address to be assigned.
- This is done at the time of variable declaration.
- A pointer that is assigned NULL is called a null pointer.

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

```
int main () {
    int *ptr = NULL;
    cout << "The value of ptr is " << ptr ;

    return 0;
}
```

- If a pointer contains the null (zero) value, it is assumed to point to nothing.



## Subject :- OOP

### Void Pointer

1. A void pointer is a pointer that has no associated data type with it. A void pointer can hold address of any type and can be type casted to any type.

```
int a = 10;  
char b = 'x';
```

```
void *p = &a; // void pointer holds address of int 'a'  
p = &b; // void pointer holds address of char 'b'
```

## Subject :- OOP

### Enum Data Type

Enumeration is a **user defined datatype** in C/C++ language. It is used to assign names to the **integral constants** which makes a program easy to read and maintain. The keyword “enum” is used to declare an enumeration.

The following is the syntax of enums.

```
enum enum_name{const1, const2, ..... };
```

Here,

**enum\_name** – Any name given by user.

**const1, const2** – These are values of type flag.

Example: enum qlty {excellent, good, average};

qlty q1;

## Subject :- OOP

### Union Data Type

Union is a user-defined datatype. All the members of union share same memory location.

Size of union is decided by the size of largest member of union.

If you want to use same memory location for two or more members, union is the best for that.

Here,

Here is the syntax of unions: -

```
union union_name {  
    member_definition;  
}union_variables;
```

- **union\_name** – Any name given to the union.
- **member definition** – Set of member variables.
- **union\_variable** – This is the object of union.

# Example:

- union stu
- {
- int roll\_no;
- char grade;
- float per;
- } st1;