Object Oriented Programming

Unit 2

Inheritance and Pointers

Major features in OOP that makes them different than non-OOP languages

- Encapsulation
- Inheritance
- Polymorphism.

Hello I am Bjarne Stroustrup.

I designed and implemented C++

Encapsulation Enforces Modularity

Inheritance Passes "Knowledge" Do

Polymorphism Takes any Shape

- What is Structure? In C?
- What is Class?
- Do you find Class is an example of Encapsulation??
- Wrapping of data and functions in a single unit
- Real world example :
- Car with petrol supported engine → can change Color, lights, visual appearance → Cant work on diesel (protection)

❑ Inheritance?

It allows the child class to acquire the properties (the data members) and functionality (the member functions) of parent class.

❑ Real world example :   child takes birth it inherit some qualities from his/her parents.

Same in the case of programming., Child Class d **class** takes some characteristics Parent Class

base class.

Polymorphism

- Real world example :milk for drinking and milk can also be used to make curd, butter etc.

- In programming , polymorphism means MANY FORMS
- To use something in different forms.

- to maintain the code and to maintain the simplicity of the code we use the concept of polymorphism. Very popular exan
  same name but the operations ar

Sum(int, int)
Sum(float, float)
Sum(int, float)

In Shopping mall behave like customer
In Bus behave like Passenger
In School behave like Student
At home behave like son

If Someone creating a game
Maths teacher, footballer, Businessman

Maths Teacher : **walk() talk()** teachesMaths()
Footballer : **walk() talk()** playsFootball()
Business: **walk() talk()** RunBusiness()

In C++, inheritance is a process in which one object acquires all the properties and behaviors of its parent object automatically.

Person(): walk() Talk()   // Reusability of code

B
MT   RB()
Tm()

F

Inheritance allows the child class to acquire the properties (the data members) and functionality (the member functions) of parent class.

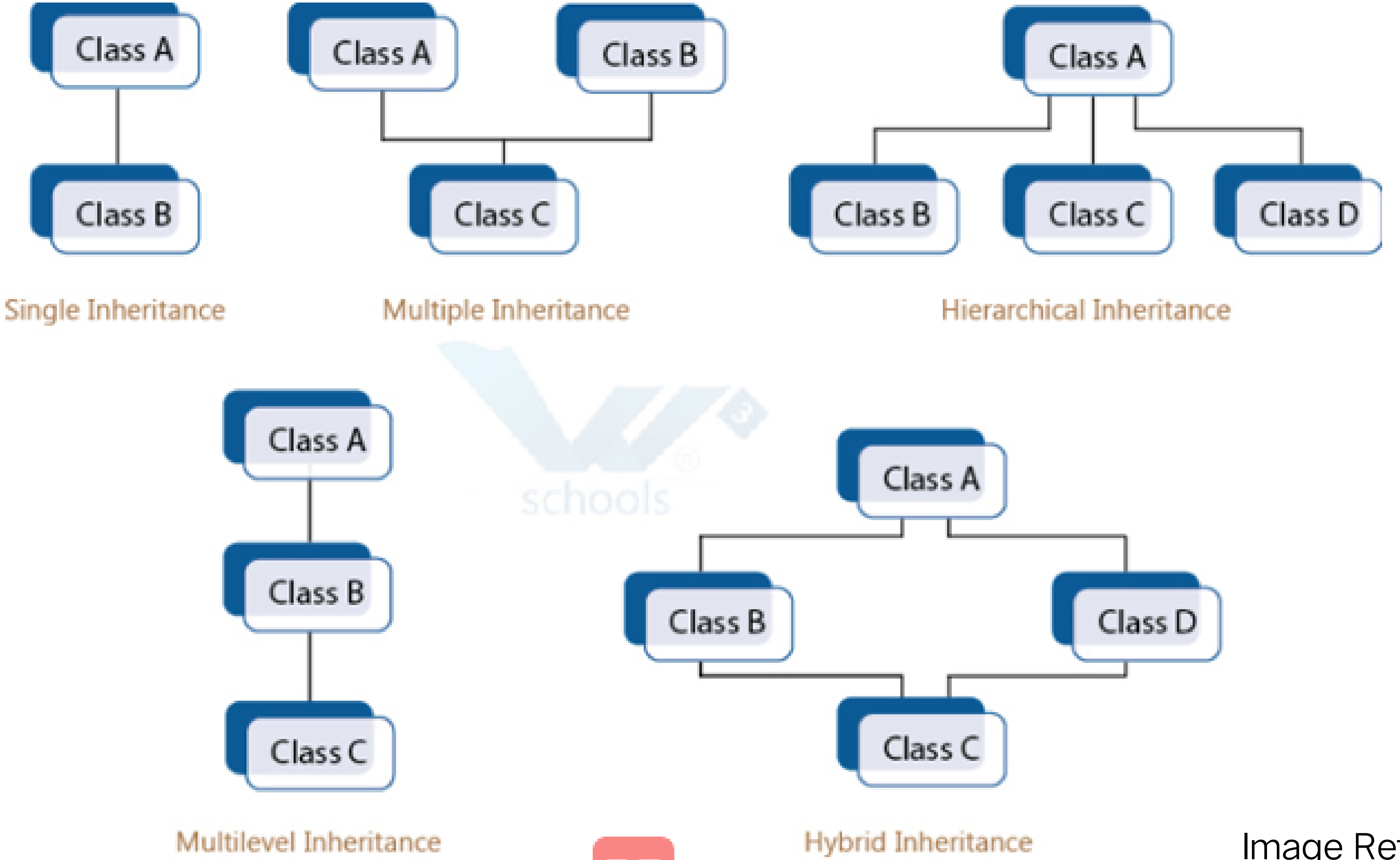
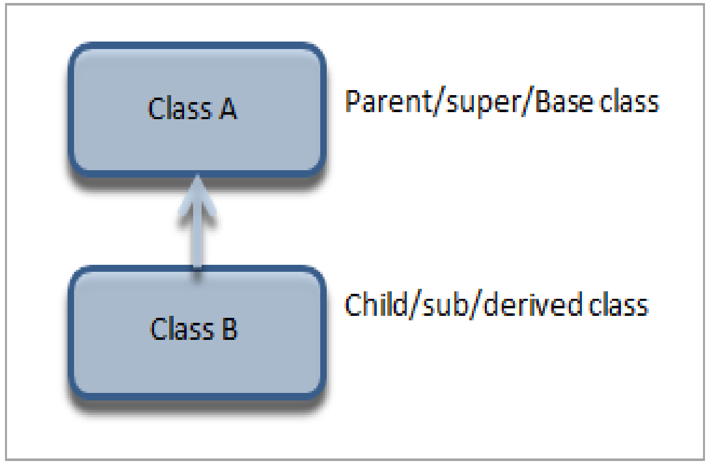


Image Ref: w3 schools

```
class derived_class_name : visibility-mode base_class_name
{
    // body of the derived class.
}
```



Examples:

If parent class is color derived can be red, black..
Shape →circle rectangular then rectangular may be further divided into square.
vehicle→car, truck,Bus
Account→ saving,current
Engg_Dept→ Entc,Comp,Mech

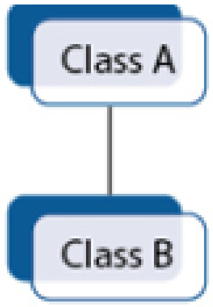
Class car : public vehicle

Access Specifier/ Access Modifier/Visibility mode

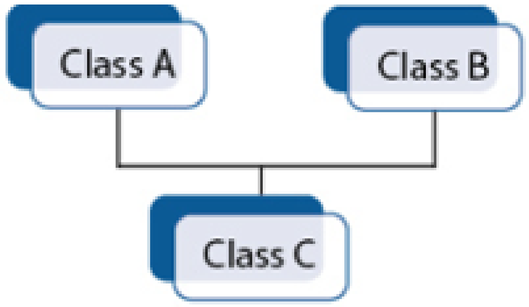
Public : all the class members declared under public will be available to everyone. The data members and member functions declared public can be accessed by other classes too.

Private : no one can **access** the class members declared private outside that class. If someone tries to access the private member, they will get a compile time error. By default class variables and member functions are private.

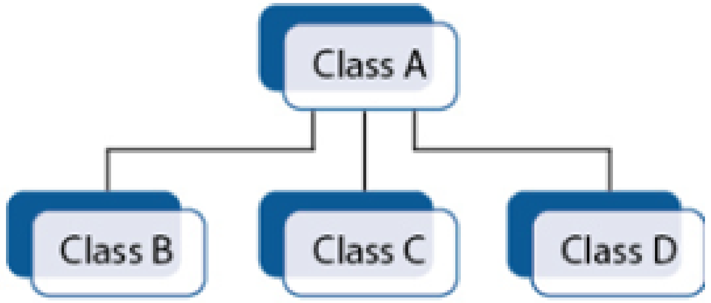
Protected : it is similar to private, it makes class member inaccessible outside the class. But they can be accessed by any subclass of that class.



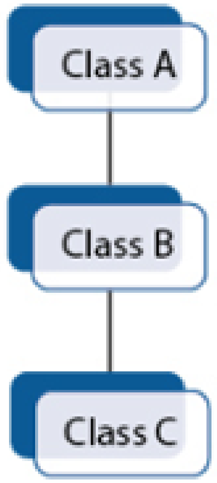
Single Inheritance



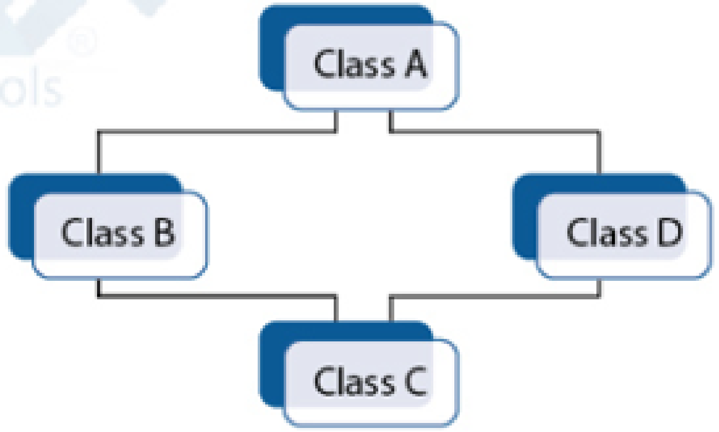
Multiple Inheritance



Hierarchical Inheritance



Multilevel Inheritance



Hybrid Inheritance

Image Ref: w3 schools

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

class Parent
{      public:
    int p;
};
class Child : public Parent
{
    public:
    int c;
};

int main()
{
Child obj1;   // An object of class child has all data members and member functions of class
parent                   obj1.c = 7;
obj1.p = 91;
cout<<"Child id is"<<obj1.c;
cout<<"Parent id is"<<obj1.p;
 return 0;
```

```
class derived_class_name :: visibility-mode base_class_name
{
    // body of the derived class.
}
```

Single

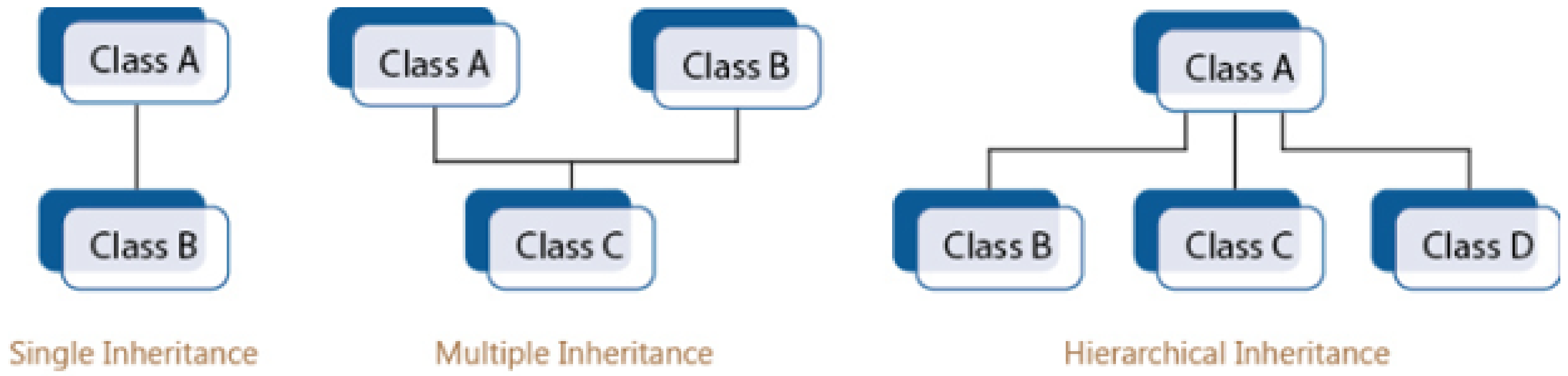
```
#include <iostream>
using namespace std;
class A {
public:
  A(){
     cout<<"Constructor of A class"<<endl;
  }
};
class B: public A                                    // B is child A is parent
{
public:
  B(){
     cout<<"Constructor of B class";
  }
};
int main() {
                                                     //Creating object of
   B obj;
   return 0;
}
```

Single

```
#include <iostream>
using namespace std;
 class Animal {
   public:
 void eat() {
    cout<<"Eating..."<<endl;
 }
   };
   class Dog: public Animal    // dog is child and animal is parent
   {
       public:
     void bark(){
    cout<<"Barking...";
     }
   };
int main(void) {
    Dog d1;
    d1.eat();
    d1.bark();
    return 0;
```

```
class derived_class_name :: visibility-
mode base_class_name
{
    // body of the derived class
}
```

Single



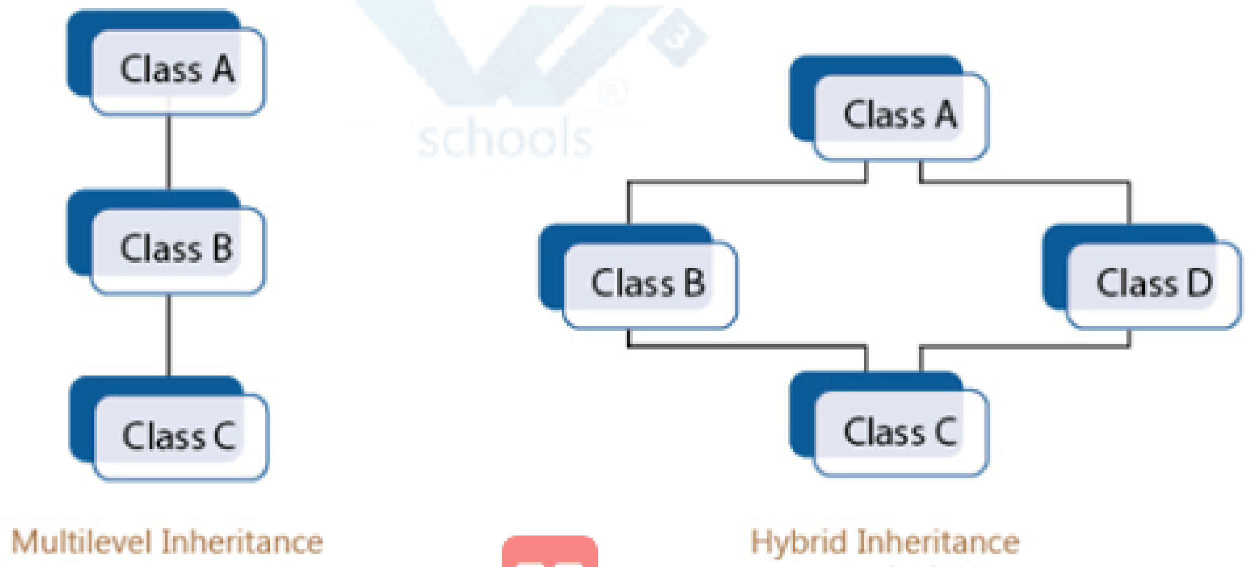


Image Ref: w3 schools

Iostream ??

Namespace??

- Header file → .h extension
- Stdio.h -> printf() declared inside it. Return type, arguments, syntax is involved.
- For definition of printf () we have library file.
- We include these files for ease of use.. Compiler should be aware of printf word so we need to include header files.

Same way

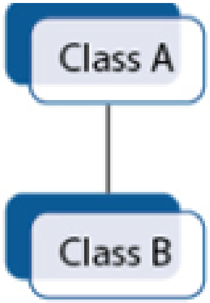
Namespace is a group of declaration.

Its not a class

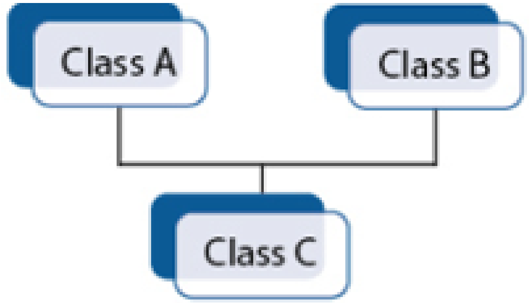
iostream (.h) (Standard input output stream) in which std lives.

Std is namespace in which cout lives.

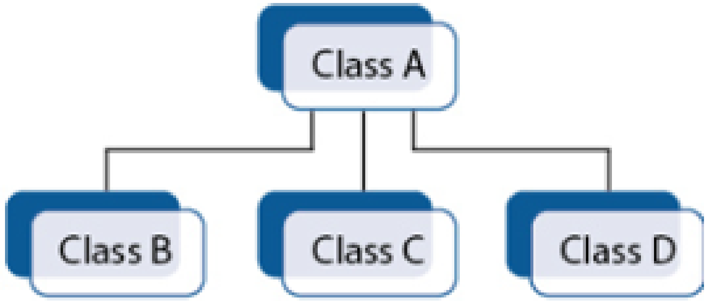
std::cout<<“hello ”;



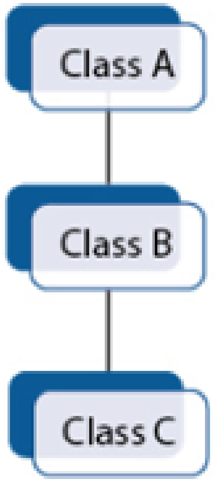
Single Inheritance



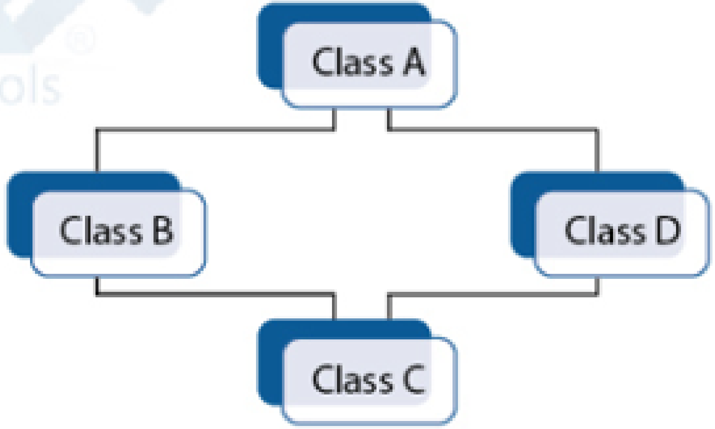
Multiple Inheritance



Hierarchical Inheritance



Multilevel Inheritance



Hybrid Inheritance

Image Ref: w3 schools

Multiple Inheritance syntax

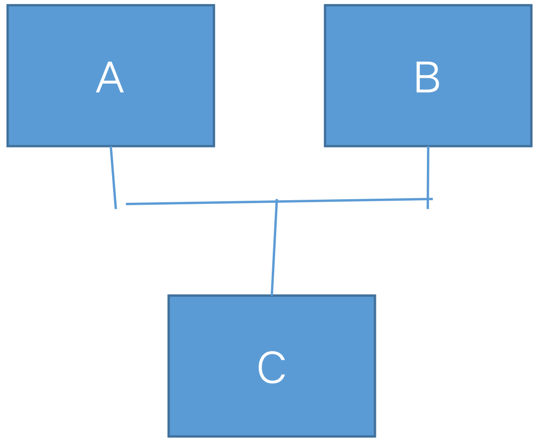
```
class A
 {
   ..........
 };

 class B
 {
   ...........
  } ;

 class C : acess_specifier A , access_specifier B
{
   ...........
 } ;
```

O/P :

```
B's constructor called
A's constructor called
C's constructor called
```

```
class A
{
public:
  A()  { cout << "A's constructor called" << endl; }
};

class B
{
public:
  B()  { cout << "B's constructor called" << endl; }
};

class C: public B, public A  // Note the order
{
public:
  C()  { cout << "C's constructor called" << endl; }
};

int main()
{
    C c;
```

```
using namespace std;
class A
{  public:
   int x;
   void getx()
   { cout << "enter value of x: "; cin >> x; }
};
class B
{  public:
   int y;
   void gety()
   {    cout << "enter value of y: "; cin >> y; }
};
class C : public A, public B   //C is derived from class A and class B
{  public:
   void sum()
   {   cout << "Sum = " << x + y; }
};
int main()
{   C obj1;
    obj1.getx();
    obj1.gety();
    obj1.sum();  return 0;
```



Multiple Inheritance

Note :These are just example programs.
You must use proper nomenclature for class names and variables and

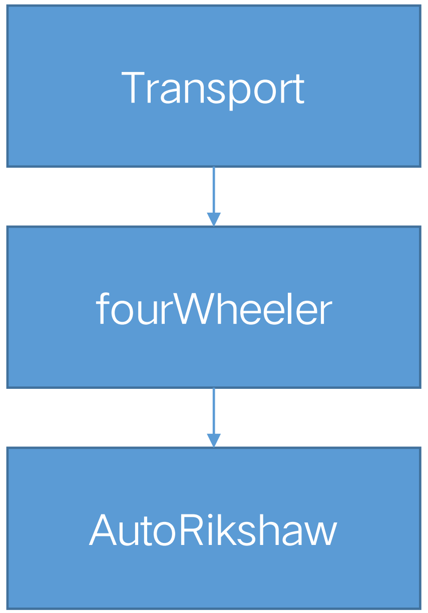
Multilevel syntax

```
class A
{
... .. ...
};
class B: public A
{
... .. ...
};
class C: public B
{
... ... ...
};
```

```
class A
{
   public:
     void display()
     {
         cout<<"Base class content.";
     }
};

class B : public A
{
};

class C : public B
{
};

int main()
{
    C obj;
    obj.display();
```

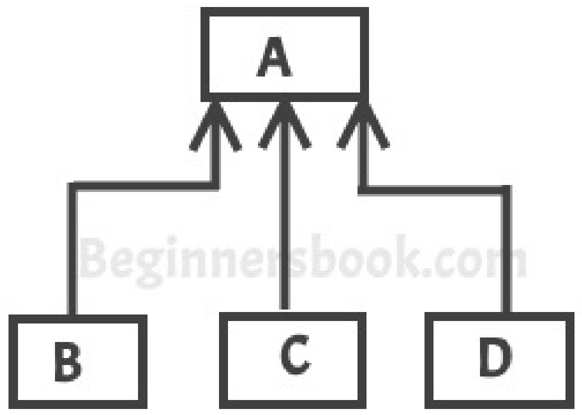
```
#include <iostream>
using namespace std;
class Transport
{
public:
Transport ()
{   cout << "This is a Transport" << endl;    }
};
class fourWheelr: public Transport
{   public:
    fourWheelr()
    { cout<<"Transport have 4 wheels as object"<<endl; }
};
class AutoRikshaw: public fourWheelr
{   public:
    AutoRikshaw()
    { cout<<"AutoRikshaw has 4 Wheels"<<endl;}
};
int main ()
{
AutoRikshaw obj;
```

Example of Multilevel



O/p :

```
Transport have 4 wheels as obj
ect

AutoRikshaw has 4 Wheels
```



Hierarchical Inheritance

```
class A // base class
{
   ..............
};
class B : access_specifier A // derived class from A
{
   ...........
 } ;
 class C : access_specifier A // derived class from A
 {
   ...........
 } ;
 class D : access_specifier A // derived class from A
 {
    ...........
 } ;
```

When more than one classes inherit a same class then this is called hierarchical inhe

```
#include <iostream>
using namespace std;
class A {
public:
  A(){
     cout<<"Constructor of A class"<<endl;
  }
};
class B: public A {
public:
  B(){
     cout<<"Constructor of B class"<<endl;
  }
};
class C: public A{
public:
  C(){
     cout<<"Constructor of C class"<<endl;
  }
};
int main() {
```

hierarchical inheritance

```
#include <iostream>
using namespace std;
class A
{  public:
   int x, y;
   void getdata()
   {   cout << "\nEnter value of x and y:\n"; cin >> x >> y; }
};
class B : public A
{  public:
              void product()
   { cout << "\nProduct= " << x * y;}
};
class C : public A
{   public:
   void sum()
```

```
int main()
{
   B obj1;
   C obj2;
   obj1.getdata();
   obj1.product();
   obj2.getdata();
   obj2.sum();
   return 0;
}
```

hierarchical inheritance

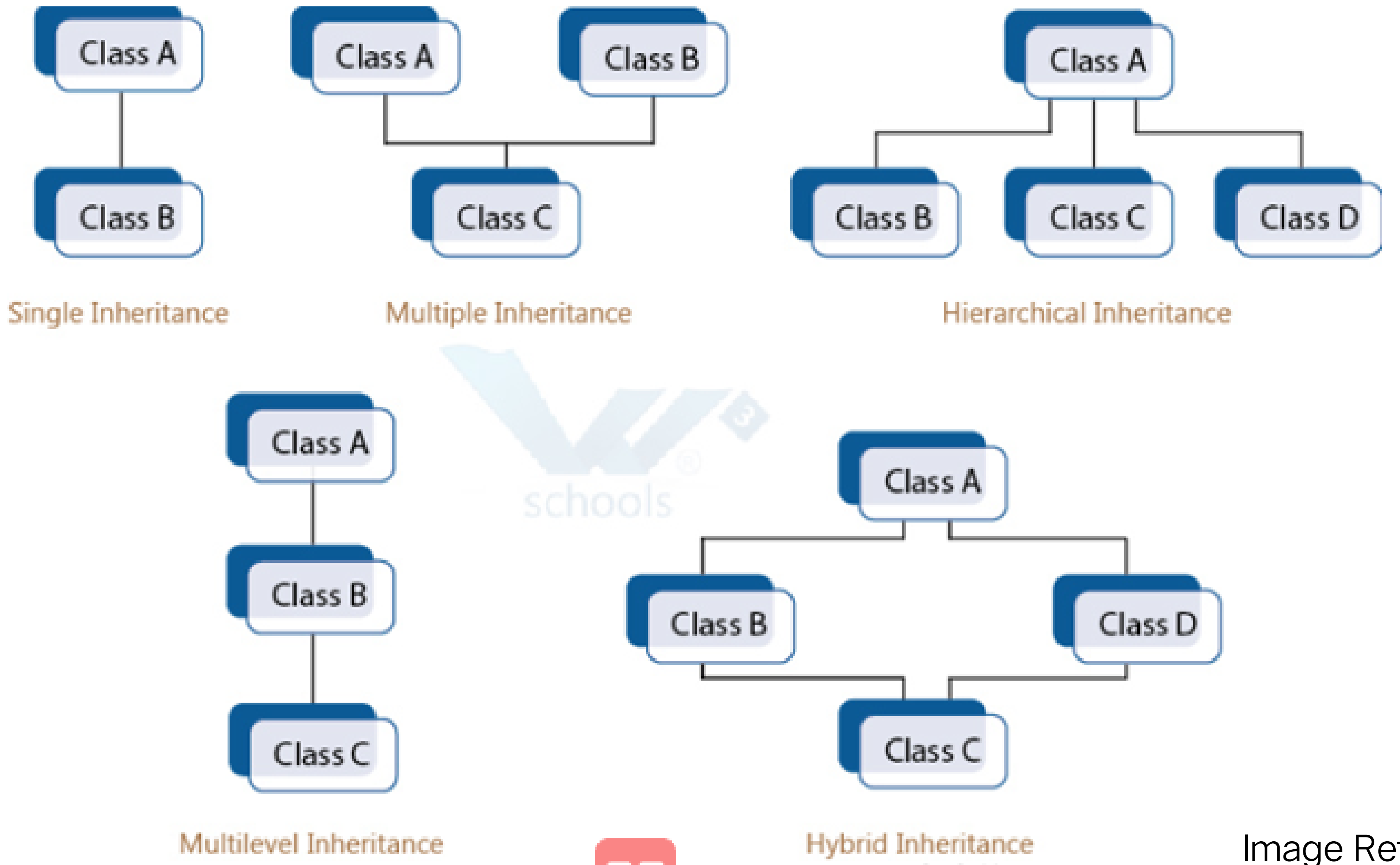
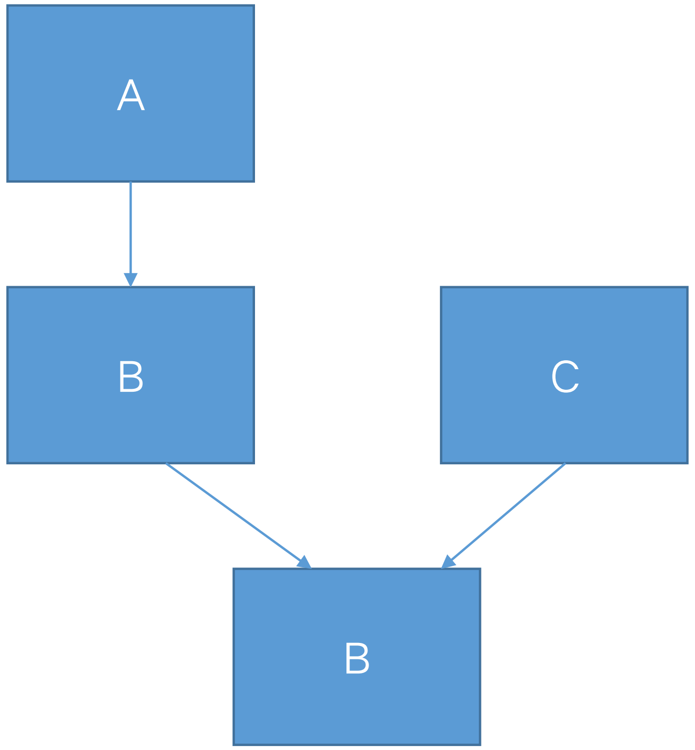


Image Ref: w3 schools

HYBRID

```
class A
{
     .........
};
class B : public A
{
     ..........
} ;
class C
{
     ...........
};
 class D : public B, public C
{
     ...........
};
```



Hybrid example

```
#include <iostream>
using namespace std;
class A
{
    public:
    int x;
};
class B : public A
{
    public:
    B()
    { x = 10; }
};
class C
 {
    public:
    int y;
    C()
    {  y = 4; }
};
```

```
class D : public B, public C
{
    public:
    void sum()
    {
        cout << "Sum= " << x + y;
    }
};

int main()
{
            D obj1
    obj1.sum();
    return 0;
}
```

```
#include<iostream>
using namespace std;
class arithmetic
{  protected:
    int num1, num2;
    public:
    void getdata()
    {
        cout<<"For Addition:";
        cout<<"\nEnter the first number: ";
        cin>>num1;
        cout<<"\nEnter the second number: ";
        cin>>num2;
    }
};
class pluss :public arithmetic
{
    protected:
    int sum;
    public:
    void add()
    { sum=num1+num2; }
```

```
class minuss
{ protected:
    int n1,n2,diff;
    public:
    void sub()
    {     cout<<"\nFor Subtraction:";
         cout<<"\nEnter the first number: ";
        cin>>n1;
        cout<<"\nEnter the second number: ";
        cin>>n2;
        diff=n1-n2;  }
};
class result:public pluss, public minuss
{
    public:
    void display()
    {
        cout<<"\nSum = "<<sum;
cout<<"\nDifference = "<<diff;
}
};
```

```
int  main()
{
    result z;
    z.getdata();
    z.add();
    z.sub();
    z.display();
    return 0;
}
```

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

class Vehicle
{
  public:
  Vehicle()
 {
cout <<"vehicle constructor" << endl;
 }
};
class Fare
{
public:
Fare()
 {
cout<<"Fare of Vehicle\n";
  }
};
class Car: public Vehicle
```

```
class Bus: public Vehicle, public
Fare
{
public:
Bus()
{ cout<<"\nhello bus";}
};


int main()
{
Bus obj2;
 return 0;
}
```

o/p

```
vehicle constructor
Fare of Vehicle

hello bus
```

```
class a
{
private:
 .........
public:
void abc() {  .... }
};

class b
{
private:
 ......
public:
void abc() { ..... }
};

class c : public a , public b
{
private: .....
public:
....
```

```
void main()

{

c obj;

obj.abc();

.................

}
```

Ambiguity in Multiple Inheritance

Solution :

obj.a :: abc()

```
#include<iostream>
using namespace std;
	class ClassA
	{
		public:
		int a;
	};
	class ClassB : public ClassA
	{
		public:
		int b;
	};
	class ClassC : public ClassA
	{
		public:
		int c;
	};
	class ClassD : public ClassB, public ClassC
	{
		public:
		int d;
	};
```

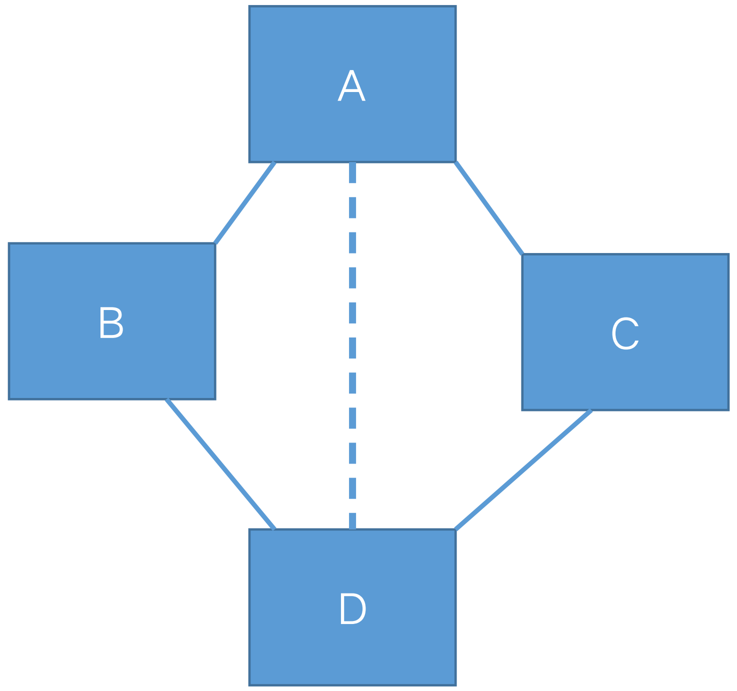
```
int main()
	{

	ClassD obj;

	//obj.a = 10;			//Statement 1, Error occur
	//obj.a = 100;			//Statement 2, Error occur

	obj.ClassB::a = 10;		//Statement 3
	obj.ClassC::a = 100;		//Statement 4

	obj.b = 20;
	obj.c = 30;
	obj.d = 40;

	cout<< "\n a through ClassB  : "<< obj.ClassB::a;
	cout<< "\n a through ClassC  : "<< obj.ClassC::a;

	cout<< "\n B : "<< obj.b;
	cout<< "\n C : "<< obj.c;
	cout<< "\n D : "<< obj.d;
```

- ✓ In the above example, both ClassB & ClassC inherit ClassA, they both have single copy of ClassA. However ClassD inherit both ClassB & ClassC, therefore ClassD have two copies of ClassA, one from ClassB and another from ClassC.

- ✓ If we need to access the data member a of ClassA through the object of ClassD, we must specify the path from which a will be accessed, whether it is from ClassB or ClassC, bco'z compiler can't differentiate between two copies of ClassA in ClassD.

There are two ways to avoid c++ ambiguity

1. Using scope resolution operator
2. Using virtual base class



To remove multiple copies of ClassA from ClassD, (A-B-D and A-C-D) we must inherit ClassA in ClassB and ClassC as virtual class.

```
#include<iostream>
using namespace std;
    class ClassA
    {
        public:
        int a;
    };

    class ClassB : virtual public ClassA
    {
        public:
        int b;
    };
    class ClassC : virtual public ClassA
    {
        public:
        int c;
    };

    class ClassD : public ClassB, public ClassC
    {
        public:
```

```
int main()
    {

  ClassD obj;

  obj.a = 10;        //Statement 3
  obj.a = 100;       //Statement 4

  obj.b = 20;
  obj.c = 30;
  obj.d = 40;

  cout<< "\n A : "<< obj.a;
  cout<< "\n B : "<< obj.b;
  cout<< "\n C : "<< obj.c;
  cout<< "\n D : "<< obj.d;
return 0;
 }
```

ClassD have only one copy of ClassA therefore statement 4 will overwrite the value of a, given in statement 3.

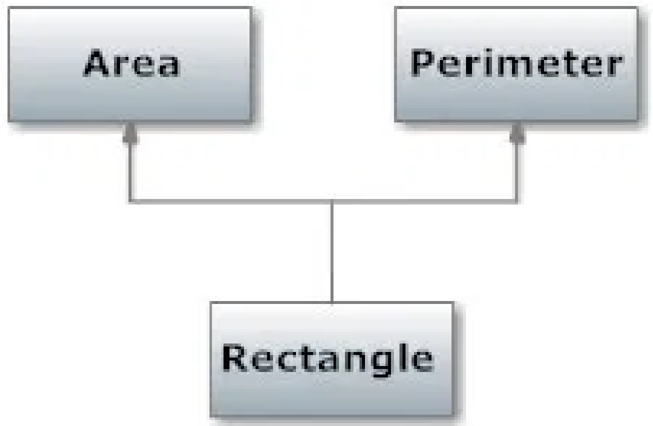
Multiple

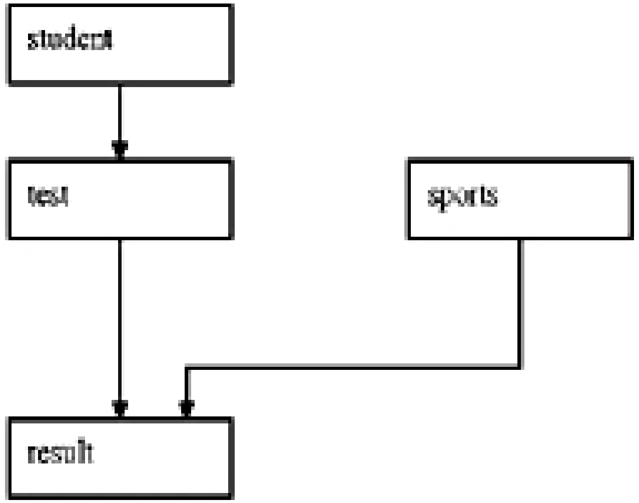
Justify : multiple base one child

Syntax:

```
class A
 {
   ..........
 };

 class B
 {
   ...........
  } ;

 class C : access_specifier A ,
access_specifier B
{
   ...........
```

```
class Area
{
};

class perimeter
{
};

class rectangle: public Area, public
Perimeter
{
};
```





Hybrid

Justify : multiple + Multiple

Write syntax of multiple and multilevel both.

```
Class student
{
};
Class test : public student
{
};

Class result: public test
{
};

Class sports
{
};
```

ACCESSBILITY	PRIVATE	PROTECTED	PUBLIC
Accessible from own class ?	yes	yes	yes
Accessible from dervied class ?	no	yes	yes
Accessible outside dervied class ?	no	no	yes

- **Pure virtual function** :

virtual functions with no definition. They start with **virtual** keyword and ends with =0

```
virtual void f() = 0;
```

Properties of Abstract Class :

- Abstract Class is a class which contains atleast one Pure Virtual function in it.
- Abstract class can have normal functions and variables along with a pure virtual function.
- Abstract classes cannot have objects.
- We can create constructors of an abstract class.
- In C++, we don't need to add an abstract keyword as the base class, which has at least one pure virtual function is understood to be an abstract class.

- If we don't override the virtual function in the derived class, then the derived class also

```
#include <iostream>
using namespace std;
class A
{
public:
  virtual void test() = 0; //declaring a virtual function
};

class B : public A
{

public:
  void test()
  {
    cout << "Hello I am the virtual function running in derived class!! :)" << endl;
  }
};

int main()
{
  B obj;
```

Have You observed both function have same name !!

Override

```
#include <iostream>
using namespace std;
class A
{
public:
  virtual void test() = 0;
};

class B : public A
{
public:
  void test()
  {
    cout << "Hello I will not run!! :(" << endl;
  }
};

int main()
{
  A obj;          //ERROR
```

- ✓ C++ abstract class is conceptually a class that cannot be instantiated and it should be implemented as a class that has one or more pure virtual (abstract) functions.
- ✓ A pure virtual function is one which must be overridden by any concrete (i.e., non-abstract) derived class.

Friend Function (concept from Unit 1)

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

class Temperature
{
int celsius;
public:
Temperature()
{
celsius = 0;
}
friend int temp( Temperature );   // declaring friend function
};

int temp( Temperature t )      // friend function definition
{
t.celsius = 40;
return t.celsius;
}
```

```
int main()
{
Temperature tm;
cout << "Temperature in celsius : " << temp( tm ) << endl;
return 0;
}
```

friend function of the class 'Temperature'. In the friend function, we directly accessed the private member celsius of the class 'Temperature'.

When the first statement of the main function created an object 'tm' of the class 'Temperature' thus calling its constructor and assigning a value 0 to its data member celsius.

The second statement of the main function called the function 'temp' which assigned a

Friend Class (concept from Unit 2)

```
class A
{
   friend class B;

};

class B
{

};
```

Class B is declared as a friend of class A
So, now all the member functions of class B became friend functions of class A.

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

class Square
{
friend class Rectangle;
int side;
public:
Square ( int s )
{
side = s;
}
};
```

```
class Rectangle
{
    int length;
    int breadth;
    public:
    int getArea()
    {
        return length * breadth;
    }
    void shape( Square a )
    {
        length = a.side;
        breadth = a.side;
```

```
int main()
{
    Square square(5);
    Rectangle rectangle;
    rectangle.shape(square);
    cout << rectangle.getArea() << endl;
    return 0;
}
```

access any private member of Square.
In the main function, the first statement created an object 'square' of the class Square, thus calling its constructor and assigning 5 to its data member side.
The second statement in the main function created an object 'rectangle' of the class Rectangle.
In the third statement, 'rectangle' called the function shape() with the object of the class Square passed as its argument.
In the 'shape()' function, the value of 'side' (private data member of Square) was assigned to length and breadth.

Nested Class example 1

```
#include<iostream>
using namespace std;
class A {
  public:
  class B {
    private:
    int num;
    public:
    void getdata(int n) {
      num = n;
    }
    void putdata() {
      cout<<"The number is "<<num;
    }
  };
};
int main() {
  A :: B obj;
  obj.getdata(9);
  obj.putdata();
```

Nested Class example 2

```
#include <iostream.h>
class Nest
{
public:
class Display
{
private:
int s;
public:
void sum( int a, int b)
{ s =a+b; }
void show( )
{ cout << "\nSum of a and b is:: " << s;}
};
};
void main()
{
Nest::Display x;
x.sum(12, 10);
x.show();
```

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

class A
{
private :
int a;
public:
friend void fun(A,B);
void setdata(int x )
{ a=x; }
};

class B
{
private:
int b;
public:
friend void fun(A,B);
void setdata(int y)
```

```
void fun(A o1,B o2)
{
cout<<"sum is "<<o1.a+o2.b;
}



int main()
{
A obj1;
B obj2;
obj1.setdata(4);
obj2.setdata(5);
fun(obj1,obj2);
return 0;
}
```

A **forward declaration** tells the compiler about the existence of an entity before actually defining the entity.

Before pointers in C++............

Check your basics of C++ programming are clear or not ??

```
/* C++ program to create a simple class and object.*/

#include  <iostream>
using namespace std;

class Hello
{
    public:
        void sayHello()
        {
                cout << "Hello World" << endl;
        }
};

int main()
{
    Hello h;

    h.sayHello();
```

```
/* C++ program to create an object of a class
and access class attributes */

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

// class definition
// "student" is a class
class Student {
public: // Access specifier
    int rollNo; // Attribute (integer variable)
    string stdName; // Attribute (string variable)
    float perc; // Attribute (float variable)
};

int main()
{
    // object creation
    Student std;

    // Accessing attributes and setting the values
    std.rollNo = 101;
    std.stdName = “Mrudul";
    std.perc = 50.2;

    // Printing the values
    cout << "Student's Roll No.: " << std.rollNo << "\n";
    cout << "Student's Name: " << std.stdName << "\n";
    cout << "Student's Percentage: " << std.perc << "\n";

    return 0;
}
```

```
/* C++ program to create multiple objects of a class */

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

// class definition
// "student" is a class
class Student {
public: // Access specifier
    int rollNo;
    string stdName;
    float perc;
};
int main()
{
    // multiple object creation
    Student std1, std2;

    // Accessing attributes and setting the values
    std1.rollNo = 101;
    std1.stdName = “Mrudul";
    std2.rollNo = 102;
    std2.stdName = “ram";
    std2.perc = 99.99;


// Printing the values
    cout << "student 1..."
         << "\n";
    cout << "Student's Roll No.: " << std1.rollNo << "\n";
    cout << "Student's Name: " << std1.stdName << "\n";
    cout << "Student's Percentage: " << std1.perc << "\n"
;

    cout << "student 2..."
         << "\n";
    cout << "Student's Roll No.: " << std2.rollNo << "\n";
    cout << "Student's Name: " << std2.stdName << "\n";
    cout << "Student's Percentage: " << std2.perc << "\n"
;

    return 0;
```

```
/* C++ program to create class methods*/

#include <iostream>
using namespace std;

// class definition
class Sample {
public:
void printText1()    // method definition 1
    {
        cout << “Hello SE students\n";
    }

    // method definition 2
    void printText2()
    {
        cout << "Let's learn together\n";
    }

    // method definition 3
    // it will accept value while calling and print it
    void printValue(int value)
```

```
int main()
{
    // creating object
    Sample obj;

    // calling methods
    obj.printText1();
    obj.printText2();
    obj.printValue(101);

    return 0;
}
```

```
/* C++ program to define a class method
    outside the class definition*/

#include <iostream>
using namespace std;

// class definition
class Sample {
public: // method declarations
    void printText1();
    void printText2();
    void printValue(int value);
};

// Method definitions outside the class
// method definition 1
void Sample::printText1()
{
    cout << "IncludeHelp.com\n";
}

// method definition 2
void Sample::printText2()
{
    cout << "Let's learn together\n";
}

// method definition 3
// it will accept value while calling and print it
void Sample::printValue(int value)
{
    cout << "value is: " << value << "\n";
}

int main()
{
Sample obj;

    // calling methods
    obj.printText1();
    obj.printText2();
    obj.printValue(101);
```

```
/*C++ program to create class for a student.*/
#include <iostream>
using namespace std;
class student
{
    private:
        char  name[30];
        int   rollNo;
        int   total;
        float perc;
    public:
        void getDetails(void);
        void putDetails(void);
};
//member function definition, outside of the class
void student::getDetails(void){
    cout << "Enter name: " ;
    cin >> name;
    cout << "Enter roll number: ";
    cin >> rollNo;
    cout << "Enter total marks outof 500: ";
```

```
void student::putDetails(void)
{
    cout << "Student details:\n";
    cout << "Name:"<< name << ",Roll Number:" <<
rollNo << ",Total:" << total << ",Percentage:" << perc;
}

int main()
{
    student std;     //object creation

    std.getDetails();
    std.putDetails();

    return 0;
}
```

Pointer

- Pointer is a variable which is used to store the address of another variable.

- Pointer are declared by * with variable name.

- Ex: int x ,*px

- Pointers are defined without *

- Pointer value accessed with *

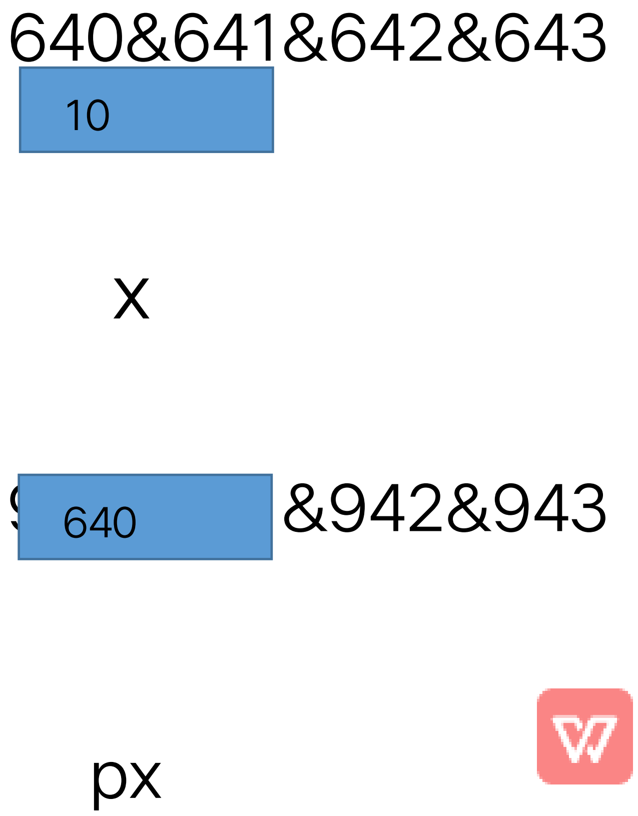
- Ex: int x=10,***px**   // declare
  px=&x   //define
  printf(“%d”,*px);

- To store the address of integer variable we require integer pointer
- To store the address of float variable we require float pointer
- To store the address of char variable are we require char pointer

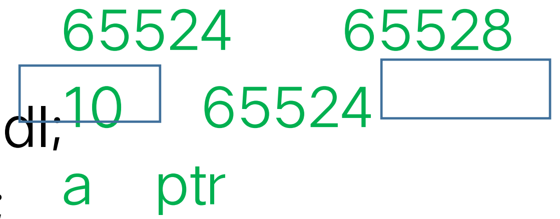
```
int *iptr ;      //creates an integer pointer iptr
char *cptr ;      //creates a character pointer
cptr
float *fptr ;      //creates a float pointer fptr
```

```
int i = 25 ;     // declares an int variable i
int *iptr ;      // declares an int pointer iptr
iptr = &i ;      // stores the memory address of i into
iptr
```

```
int x, *px    //Declare
x=10;       //Define
px=&x;     //Define
```



```
#include<iostream>
using namespace std;
int main()
{
 int a = 10, *ptr ;
ptr = &a ;
cout<<"Address of variable a =  "<< ptr <<endl;
cout<<"Value of variable a =  "<< *ptr<<endl;
cout<<"Address of variable ptr = "<< &ptr;
return 0;
}
```



o/p:

```
Address of variable a = 65524
Value of variable a = 10
Address of variable ptr =
#include <iostream>
using namespace std;

int main ()
{
  int firstvalue = 5, secondvalue = 15;
  int * p1, * p2;

  p1 = &firstvalue;  // p1 = address of firstvalue

  p2 = &secondvalue; // p2 = address of secondvalue

  *p1 = 10;          // value pointed to by p1 = 10

   *p1 = 20;          // value pointed to by p1 = 20


  cout << "firstvalue is " << firstvalue << '\n';
  cout << "secondvalue is " << secondvalue << '\n';
```

```
#include <iostream>
using namespace std;
int main() {
   int n = 5;
   int *pn = &n;
   int **ppn = &pn;

   cout  << "Value of n:\n"
         << "direct value: " << n << endl
         << "indirect value: " << *pn << endl
         << "doubly indirect value: " << **ppn << endl
         << "address of n: " << pn << endl
         << "address of n via indirection: " << *ppn << endl;
}
```

[Slide 77]

Null pointer :

We can initialize a pointer with pointer value 0 or NULL. Its meaning is the pointer is pointing to nothing. It is called a null pointer.

```
int *p= NULL;
```

Void pointer :

In C and C++ is a generic pointer that can point to any data types e.g. int, float and char etc.

in other words, void pointer – void * – is a pointer that points to some data location in storage, which doesn’t have any specific type.

Void pointer is also known as *generic pointer in c* and c++ programs.

```
void *ptr
```

HW :
Why to use Null pointer

Array in C++

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

int main(){
   int arr[] = {11, 22, 33, 44, 55};
   cout<<arr[0]<<endl;
   cout<<arr[1]<<endl;
   cout<<arr[2]<<endl;
   cout<<arr[3]<<endl;
   cout<<arr[4]<<endl;
   return 0;
}
```

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

int main() {
    int numbers[5];

    cout << "Enter 5 numbers: " << endl;

    //  store input from user to array
    for (int i = 0; i < 5; ++i) {
        cin >> numbers[i];
    }

    cout << "The numbers are: ";

    //  print array elements
    for (int n = 0; n < 5; ++n) {
        cout << numbers[n] << " ";
    }
```

```
#include <iostream>
using namespace std;
int main(){
   //Pointer declaration
   int *p;
   //Array declaration
   int arr[]={1, 2, 3, 4};
   //Assignment
   p = arr;
   for(int i=0; i<4;i++){
     cout<<*p<<endl;
     //++ moves the pointer to next int
position
     p++;
   }
```

Slide 75

new and delete Operators

- new is used to allocate memory for a variable, object, array, array of objects.. etc **at run time**.
- "To declare memory at run time = dynamic memory allocation."
- When we use dynamic allocation? When we are not aware in advance that how much memory space is needed?

pointer_variable = new data_type;

int *age= new int

Here, pointer_variable is a variable that store the starting address of allocating memory space

```
#include <iostream.>
Using namespace std;
int main()
{
    int  *age=new int;
    char *gender=new char;
    *age=0;         // assigning default value (not necessary)
    *gender='M';    // assigning default value (not necessary)

    cout<<"Enter your age ? :";
    cin>> (*age);

    cout<<"AGE IS : "<< *age <<",GENDER IS : "<< *gender << endl;

    return 0;
}
```

- delete operator is used to Deallocates the dynamically allocated memory.

- Since the necessity of dynamic memory is usually limited to specific moments within a program, once it is no longer needed it should be freed so that the memory becomes available again for other requests of dynamic memory.

1. delete pointer_variable;
2. delete []pointer_variable;

The first expression should be used to delete memory allocated for a single element, and the second one for memory allocated for arrays of elements.

delete []array;

```
#include <iostream>
using namespace std
int main()
{
int size;  // variable declaration
int *arr = new int[size];   // creating an array
cout<<"Enter the size of the array : ";
std::cin >> size;    //
cout<<"\nEnter the element : ";
for(int i=0;i<size;i++)   // for loop
{
cin>>arr[i];
}
cout<<"\nThe elements that you have entered are :";
for(int i=0;i<size;i++)    // for loop
{
cout<<arr[i]<<",";
}
delete []arr;  // deleting an existing array.
return 0;
```

```
#include <iostream>

using namespace std;

int main()
{
    int length, sum = 0;
    cout << "Enter the number of students in the group" << endl;
    cin >> length;
    int *marks = new int[length];
    cout << "Enter the marks of the students" << endl;
    for( int i = 0; i < length; i++ )          // entering marks of students
    {
        cin >> *(marks+i);
    }
    for( int i = 0; i < length; i++ )          // calculating sum
    {
        sum += *(marks+i);
    }
    cout << "sum is " << sum << endl;
    delete[] marks;
```

If you allocate memory using new, it remains allocated until the program exits unless you explicitly deallocate with delete. The program contains only one function so memory will be deallocated after program exits, but we have used delete as it's a good programming practice to deallocate memory that isn't required further in the program.

Using The “new” Operator With Arrays (students doubt)

another use of the “new” operator is allocating memory for arrays. Here we specify the number of elements to be allocated for the array.

An Example of allocating array elements using “new” operator is given below:

```
int* myarray = NULL;
myarray = new int[10];
```

Here, new operator allocates 10 continuous elements of type integer to the pointer variable myarray and returns the pointer to the first element of myarray.

the below programming Example shows the usage of new and delete operators with array in C++. (students doubt)

```
// Example program
#include <iostream>
#include <string>
using namespace std;

int main()
 {
   int *ptr = NULL;
   ptr = new int();
   int *var = new int(12);

   if(!ptr)
   {
        cout<<"bad memory allocation"<<endl;
    }
   else
  {
       cout<<"memory allocated successfully"<<endl;
```

```
double *myarray = NULL;
myarray = new double[10];
if(!myarray)
 {cout<<"memory not allocated"<<endl;}
else
   {
     for(int i=0;i<10;i++)
       myarray[i] = i+1;
       cout<<"myarray values : ";
     for(int i=0;i<10;i++)
       cout<<myarray[i]<<"\t";
   }
delete ptr;
delete var;
delete[] myarray;

return 0;
```

Difference between Arrays and pointers

- An array is a collection of elements of similar data type whereas the pointer is a variable that stores the address of another variable.
- An array size decides the number of variables it can store whereas; a pointer variable can store the address of only one variable in it.
- Arrays can be initialized at the definition, while pointers cannot be initialized at the definition.
- Arrays are static in nature which means once the size of the array is declared, it cannot be resized according to users requirement. Whereas pointers are dynamic in nature, which means the memory allocated can be resized later at any point in time.
- Arrays are allocated at compile time while pointers are allocated at runtime.

1) the sizeof operator
sizeof(array) returns the amount of memory used by all elements in array
sizeof(pointer) only returns the amount of memory used by the pointer variable itself

2) the & operator
&array is an alias for &array[0] and returns the address of the first element in array
&pointer returns the address of pointer

3) a string literal initialization of a character array
char array[] = “abc” sets the first four elements in array to ‘a’, ‘b’, ‘c’, and ‘\0’
char *pointer = “abc” sets pointer to the address of the “abc” string

4) Pointer variable can be assigned a value whereas array variable cannot be.

```
int a[10];
int *p;
p=a; /*legal*/
a=p; /*illegal*/
```

5) Arithmetic on pointer variable is allowed.

This Pointer
Using Pointers with Objects

- Every variable that holds the address of another variable : pointer
- Objects can also have an address and there is a pointer which can point to address of an object :- THIS pointer.

- Every object in C++ has access to its own address through an important pointer called **this** pointer.

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

This operator

int pointer can storage int values

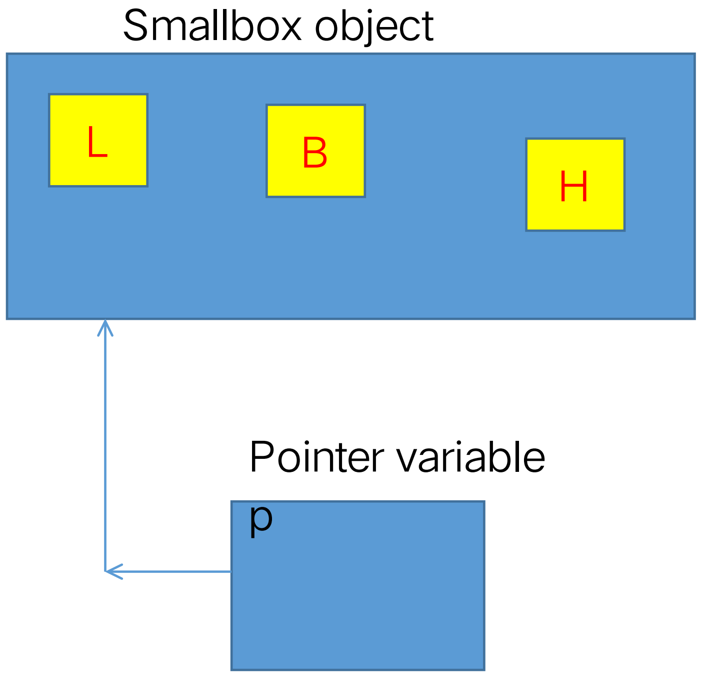
Float pointer can storage float pointer

A pointer contains address of an object is called as object pointer

```
#include<iostream.h>
#include<conio.h>
class box
{
private:
int l,b,h;
public:
void setdimension(int x,int y,int z)
{
l=x;b=y;h=z;
}
void showdimension()
{
cout<<"l="<<l<<"b="<<b<<"h="<<h;
```

```
void main()
{
box smallbox;
smallbox.setdimension(3,4,5);
smallbox.showdimension();
getch();
}
```

If we create a pointer and put address of smallbox in that pointer then what will happen??



pointer variable uses ARROW to access anything.

```
#include<iostream.h>
#include<conio.h>
class box
{
private:
int l,b,h;
public:
void setdimension(int x,int y,int z)
{
l=x;b=y;h=z;
}
void showdimension()
{
cout<<"l="<<l<<"b="<<b<<"h="<<h;
}
}
```

```
void main()
{
box  *p, smallbox;
p=&smallbox
//smallbox.setdimension(3,4,5);
//smallbox.showdimension();
P->setdimenstion(3,4,5)
P->showdimension();
getch();
}
```

This is object pointer

```
#include <iostream>
 using namespace std;
class box
{private:
int l,b,h;
public:
void setdimension (int l,int b,int h)
{
l=l;b=b;h=h; //left side is instance variable
}
void showdimension()
{
cout<<"l="<<l<<"b="<<b<<"h="<<h;
}
};
int main()
{
box  smallbox;
smallbox.setdimension(3,4,5);//instance variable
smallbox.showdimension();
return 0;
```

Will this program run properly ????

```
#include <iostream>
 using namespace std;
class box
{private:
int l,b,h;
public:
void setdimension (int l,int b,int h)
{
this->l=l;this->b=b;this->h=h;
}
void showdimension()
{
cout<<"l="<<l<<"b="<<b<<"h="<<h;
}
};
int main()
{
box  smallbox;
smallbox.setdimension(3,4,5);//instance variable
smallbox.showdimension();
return 0;
```

This pointer will be pointing to caller object.
Caller object in the program is smallbox

Caller object's l = local variable l

In previous problem we had NAME CONFLICT which we have solved by using THIS pointer

Arrays of Pointers

[61 slide]

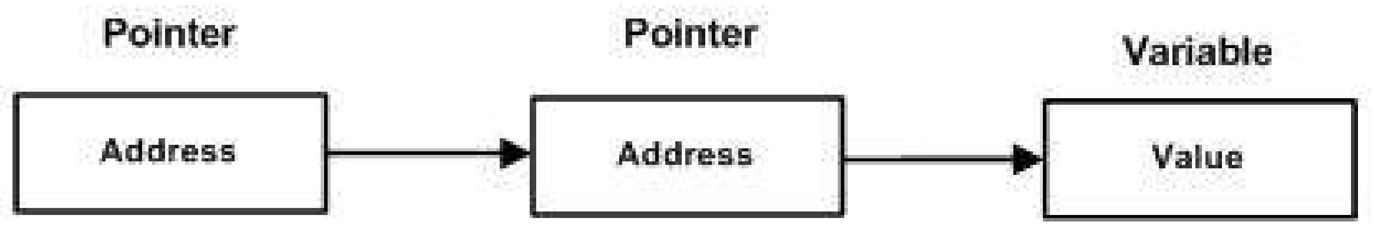
```
#include <iostream>

using namespace std;
const int MAX = 3;

int main () {
  int  var[MAX] = {10, 100, 200};
  int *ptr[MAX];

  for (int i = 0; i < MAX; i++) {
    ptr[i] = &var[i]; // assign the address of integer.
  }

  for (int i = 0; i < MAX; i++) {
    cout << "Value of var[" << i << "] = ";
    cout << *ptr[i] << endl;
  }

  return 0;
```

This declares ptr as an array of MAX integer pointers. Thus, each element in ptr, now holds a pointer to an int value. This program makes use of three integers which will be stored in an array of pointers

```
#include <iostream>
using namespace std;
int main()
{
  char *names[5] = {"john",
                  "Peter",
                  "Marco",
                  "Devin",
                  "Ronan"};
for(int i=0;i<5;i++)
    {
        std::cout << names[i] << std::endl;
    }
    return 0;
}
```

Pointers to Pointers Multiple Indirection

- 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.



Slide 57

pointer to a pointer of type int −

```
int **var;
```

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

int main () {
   int  var;
   int  *ptr;
   int  **pptr;

   var = 3000;

   // take the address of var
   ptr = &var;

   // take the address of ptr using address of operator &
   pptr = &ptr;

   // take the value using pptr
   cout << "Value of var :" << var << endl;
   cout << "Value available at *ptr :" << *ptr << endl;
   cout << "Value available at **pptr :" << **pptr << endl;
```

o/p:

```
Value of var :3000
Value available at *ptr :3000
Value available at **pptr :
3000
```

Passing pointers to functions

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

// function declaration:
double getAverage(int *arr, int size);

int main () {
   // an int array with 5 elements.
   int balance[5] = {1000, 2, 3, 17, 50};
   double avg;

   // pass pointer to the array as an argument.
   avg = getAverage( balance, 5 ) ;

   // output the returned value
   cout << "Average value is: " << avg << endl;

   return 0;
}
```

```
double getAverage(int *arr, int size) {
   int i, sum = 0;
   double avg;

   for (i = 0; i < size; ++i) {
      sum += arr[i];
   }
   avg = double(sum) / size;

   return avg;
}
```

End of unit 2 !!!!!!!!!!