DATE.....................

SUBMMITTED BY:

"SADAF SALEEM"

#2929

CS-2ND (M)

CSI-302

SUBMMITTED TO:

"SIR KHURRAM SHAHZAD Sb."

GC UNIVERSITY FSD.

DATE.....................

ASSIGNMENT #01

DATE.........................

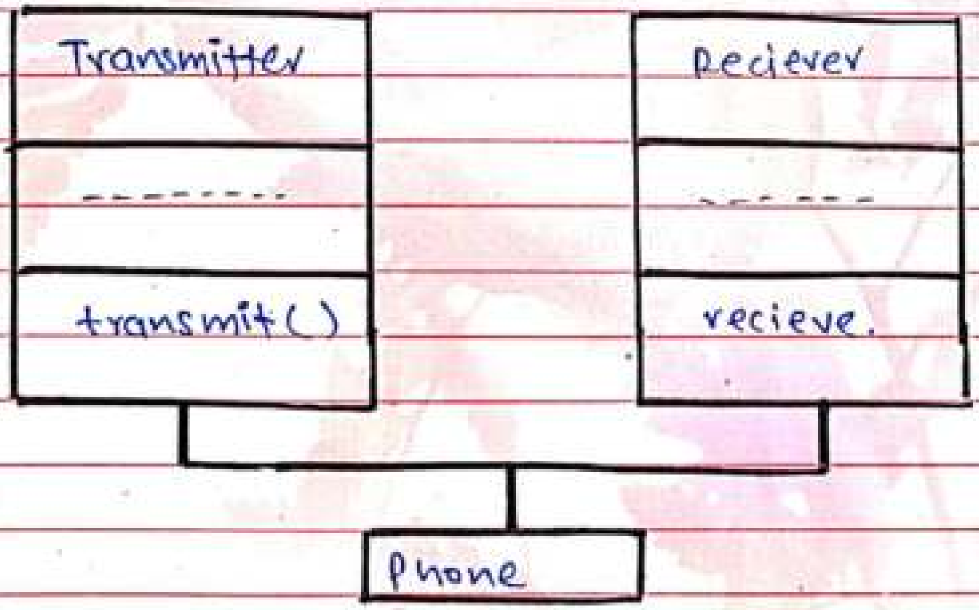
QUESTION NO. 01
"INHERITANCE"

Multi-level and Multiple inheritance:

Introduction:

1. A class can inherit more than one classes.
2. Inherited class would have properties but of all its base classes.
3. Base classes kept unchanged by this process.

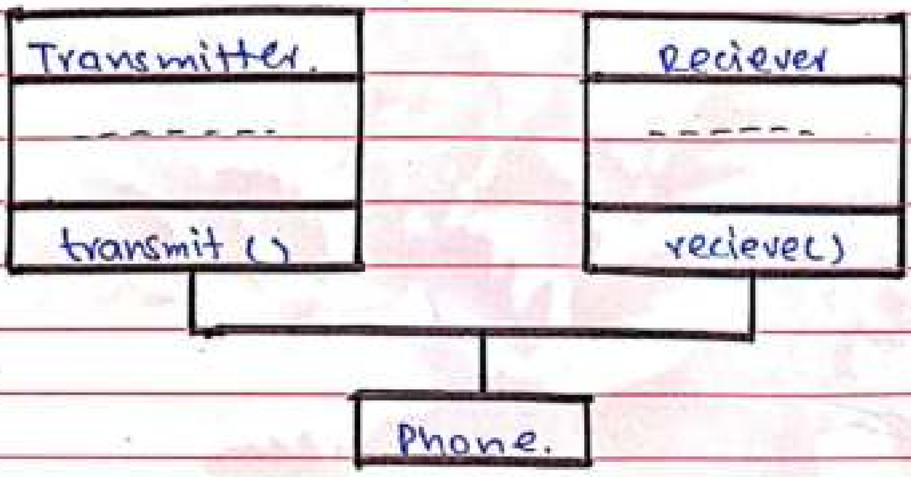
Example:



DATE.....................

Here "phone" class is inherited from both "transmitter" and "Reciever" classes.

Example syntax of C++:



```
class Transmitter {......};
class Reciever {.....};
class Phone : Public Transmitter, Public Reciever {--};
```

Example Program

```
  #include <iostream.h>
   class transmitter {
Public :
      void transmit () {
      cout << "Recieving" << endl; }
      };
   void main () {
   Phone myPhone;
```

```
myPhone.transmit();   // Transmitting
myPhone.recieve();    // Recieving.
```

Advantages of Multiple Inheritance:

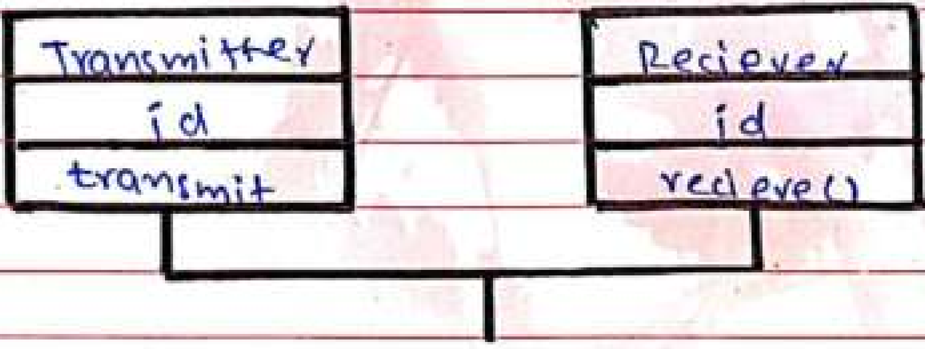
Features of more than one classes can be used into a single class.
code duplication can be avoided.

Problems in Multiple Inheritance:
Ambiguity.
Diamond shape problem.

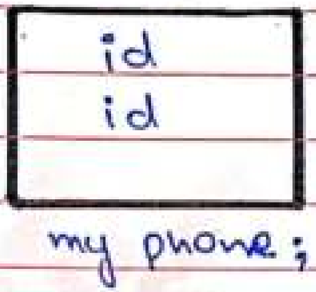
Ambiguity in Multiple Inheritance:

If a same member is coming from more than one base class then it can create ambiguity when using it in child class.

Example:



Phone my phone;



Example Program:

```
      class Transmitter {
Protected:
      int id;
Public:
      void transmit() {
      cout<<"Recieving" << endl; }
      };

Class phone : Public Transmitter, public Reciever{
Public:
      void print ID() {
      cout << "printing to ID in phone class :"<<id <<
                                          endl;
      }
    };
```

DATE.....................

Error:- id is ambiguous.

Disambiguation:

Solution of Ambiguity in Multiple Inheritance.

```
class Transmitter {
Protected:
        int id;
Public:
      void transmit () {
      cout<<"Transmitting" <<endl; }.
      };

class Reciever {
Protected:
        int id;
Public:
      void reciere () {
      cout <<"Recieving" <<endl; }
      };

   class Phone : Public Transmitter, public Reciever {
Public:
      void printID () {
```

DATE.......................

```
    cout<<"Printing ID in phone class:"<<endl; <<
    "ID from Transmitter:" << Transmitter:: id <<
     endl << "ID from Reciever:" << Reciever ::
    id << endl;
    }
    };
```

Scope Resolution operator
also called "Disambigution
operator".

Ambiguity in Multiple Inheritance

Example Program # 02 :-

```
      class Transmitter
Protected:
      int id;
Public:
      void transmit () {
      cout<<"Transmitting" << endl; }
      };

      class Reciever {
Protected:
      int id;
Public:
    void recieve() {
    cout<<"Recieving" <<endl;}
    void printID () {
    cout << "Reciever ID: "<<id<<endl;}
 };
 class Phone: Public Transmitter;Public Reciever {
  };

 void main() {
    Phone my Phone;
    myPhone.printID();
```

Error:-PrintID is ambiguous.

Disambiguition:

Example Program #02:-

```
   class Transmitter {
Protected:
      int id;
Public:
      void transmit () {
      cout << "Transmitting" <<endl;}
      void printID () {
      cout << "Transmitter ID:" << id << endl;}
    };
```

DATE......................

```
    class Reciever {
Protected:
        int id;
Public:
     void recieve() {
          cout<<"Recieving"<<endl; }
     void printID() {
          cout<<"Reciever ID: "id<<endl; }
     };

class Phone : Public Transmitter , Public Reciever {
     };

void main () {                        Disambiguation
     Phone myPhone;
     myPhone.Transmitter :: PrintID();
     myPhone.Reciever:: printID ();
  }.

// PrintID() of Transmitter class

// printID() of Reciever class
```

DATE......................

QUESTION NO. 02
"POINTERS"

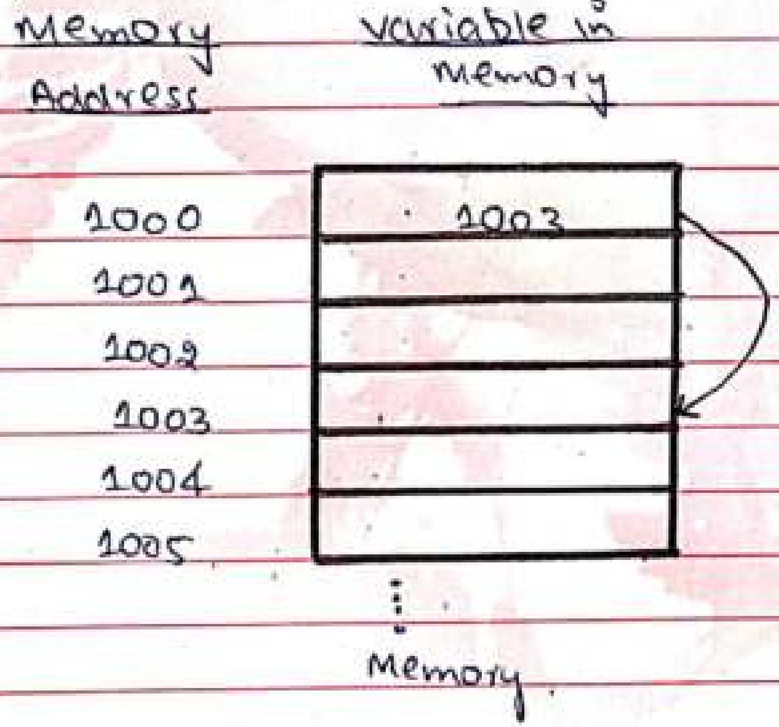
Part - 1

What are Pointers?

A pointer is a variable that holds a memory address.

This address is the location of another object. (Typically another variable) in memory.

Example:



DATE......................

Pointer Variables:

If a variable is going to hold a pointer, it must be declared as such.

A pointer declaration consist s of :

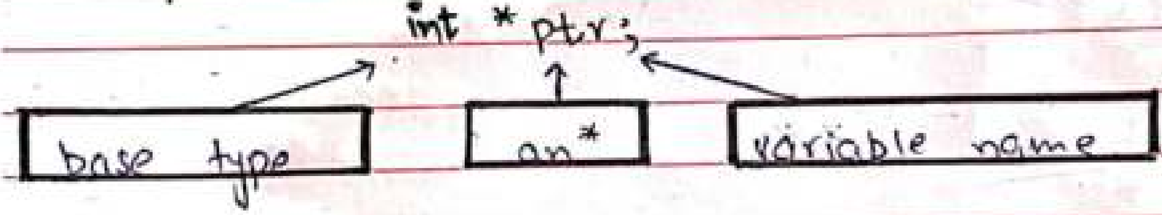
a base type

an *

and the variable name

i.e; type *name;

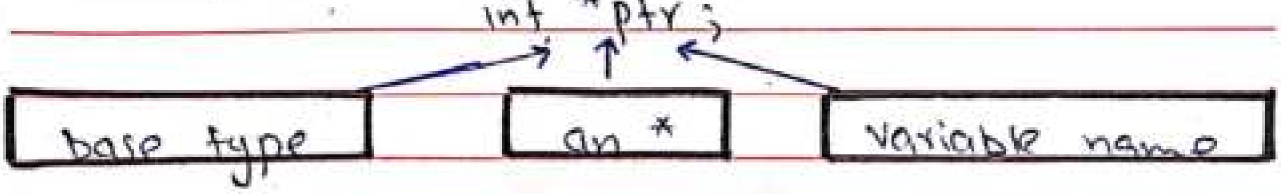
Example;



Pointers Variables cont.

- The base type of the pointer defines what type of variables the pointer can point to.
- technically, any type of pointer can ~~"type"~~ point wherever in the memory.
- All pointer arithmetic is done relative to its base type.

Example :



The Pointer operators :

1- There are two special pointer operator.
   (i)- &
   (ii)- *

2- The " & " is a unary operator that returns the memory address to its operand.

Example :

m = & count;

places into 'm' the memory address of 'count' variable. It has nothing to do with the value of count.

The Pointer operators cont.

⇒ There are two special pointer operators.
   (i)- &
   (ii)- *

- Operator '*' is the compliment of '&'.

- It is unary operator that returns the value located at the address that follows.

Example:

If m contains the memory address of count

q = *m; places the value of count into q

=> There are two special Pointer operators:

(i). &

(ii). *

- Both '&' and '*' have a higher precedence than all other arithmetic operators except the unary minus, with which they are equal.

Pointer Expressions:

As with any variable, you may use a pointer on the right-hand side of an assignment statement to assign its value to another pointer.

Example:

```
    # include<stdio.h>
```

```
int main(void)
{   int x;
    int *p1, *p2;
    p1 = &x;
    p2 = p1;
    count << p2;
    return 0;
}.

  /*print the address of x, not x's
               value! */
```

Both P1 and P2 now point to x

Pointer Arithmetic :

x ~~As with any variable, you may use a pointer on the right hand~~ x

There are only two arithmetic operations that you may use on pointers :

(i). Addition

(ii). Subtraction.

Let p1 be an integer pointer with a current value of 2000.

p1++; causes p1 to have the value 2002. The reason for this is that each time p1 is incremented, it will point to the next integer. Assume integers are 2 bytes long.

Pointer Arithmetic cont.

=> There are only two arithmetic operations that you may use on pointers:

(i)_ Addition.

(ii)_ Subtraction.

Example:

Let p1 be an integer pointer with a current value of 2000.

p1--; cause p1 to have the value 1998. The same reason is for subtraction. Assume integers are 2 bytes long.

=> There are only two arithmetic operations that you may use on pointers:

(i). Addition.

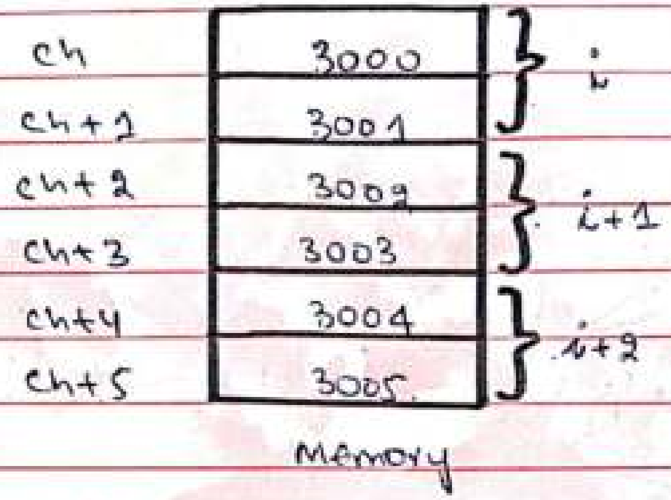
(ii)_ Substraction.

When applied to character pointers, this will appear as "normal" arithmetic because character are 1 byte long.

DATE.......................

Example:

```
char*ch=(char *)3000;
 int* i=(int *) 3000;
```



=> There are only two arithmetic operatations that you may use on pointers:

(i)_ Addition.

(ii)_ Subtraction.

You are not limited to the increment and decrement operators.

Example:

You may add or subtract integers to or from pointers.

The Expression p1 = p1 + 12;

makes p1 point to the twelfth element of p1's type beyond the one it currently points to.

DATE.......................

=> There are only two arithmetic operations that you may use on pointers:

(i)_ Addition.

(ii)_ subtraction.

- You may subtract one pointer from another in order to find the number of objects of their base type that separate the two.

- All other arithmetic operations are pohibited:

(i)_ you may not multiply or divide pointers.

(ii)_ you may not add two pointers.

(iii)_ you may not apply the bitwise operators to them.

Part-2

Pointers and Arrays:

There is a close relationship b/w pointers and arrays.

Example:

```
        Char str[80], *p1;
            p1 = str;
```

Here, p1 has been set to the address of the first array element in 'str'

To access the fifth element in 'str', you could write: OR

```
str [ 4 ]          *(p1 + 4)
```

Both statements will return fifth element.

Arrays of pointers:

Pointers may be arrayed like any other data type.

The declaration for an int pointer array of size 10 is:

```
int * x [10];
```

To assign the address of an integer variable called var to the third element of the pointer array, write

```
x [ 2 ] = & var ;
```

To find the value of 'var', write *x[2].

Passing to Functions:

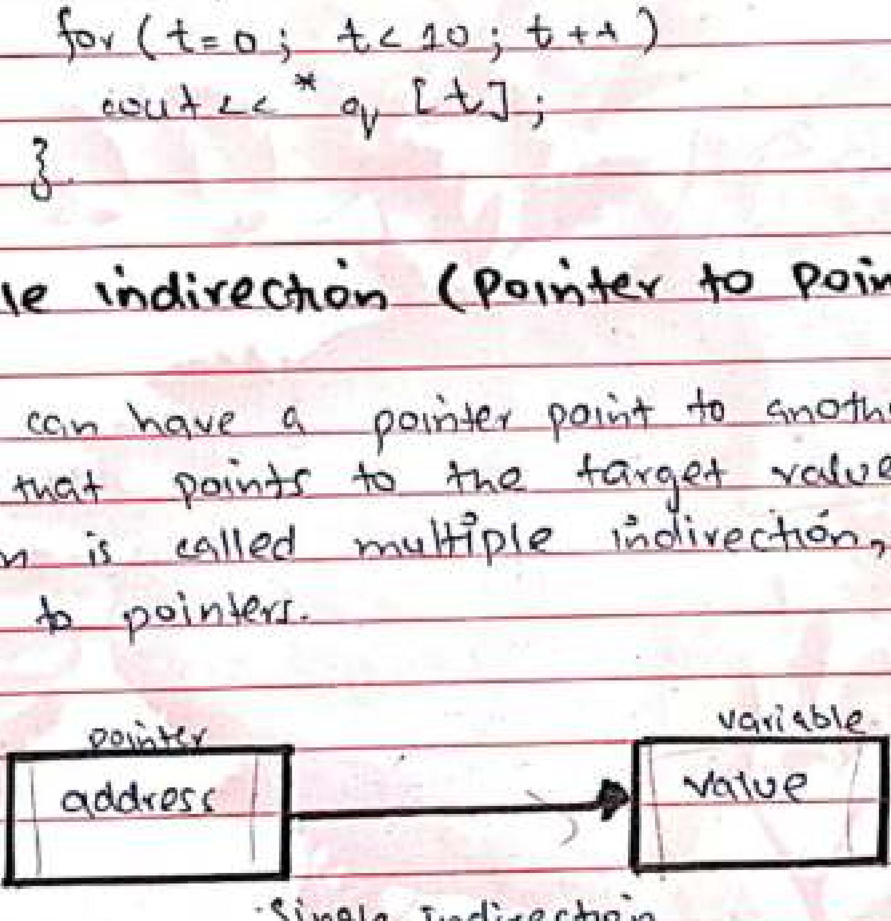
If you want to pass an array of pointers into a function, you can use the same method that you use to pass other arrays.

A function that can recieve array 'x' looks like this:

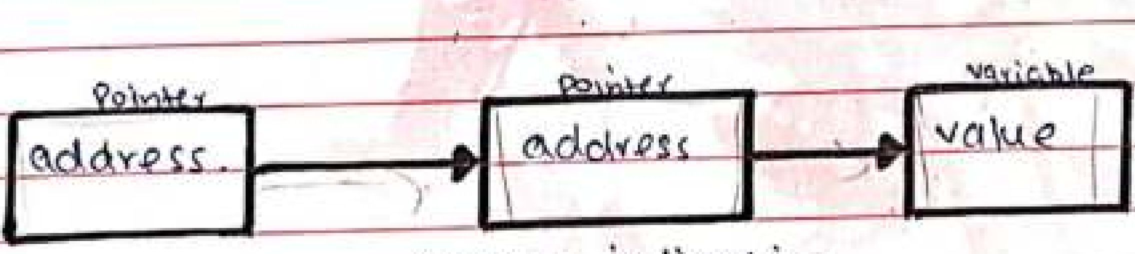
```
void display_array (int *q[])
{
   int t;
   for(t=0; t<10; t++)
     cout<< *q [t];
}.
```

Multiple indirection (Pointer to Pointer)

- You can have a pointer point to another pointer that points to the target value. This situation is called multiple indirection, or pointers to pointers.



Single Indirection



Multiple indirection

- In the case of a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the object that contains the value desired.

Multiple indirection (Pointer To Pointer) cont.

- To access the target value indirectly pointed to by a pointer to a pointer, you must apply the asterisk operator twice.

Example:

```
        int x, *p, **q;
            x=10;
            p=&x;
            q=&p;
           cout<< **q; /*print value of x.*/
```

Here, p is declared as a pointer to an integer and q as a pointer to a pointer to an integer.

DATE.....................

Initializing Pointers:

A pointer that does not currently point to a valid memory location is given the value null (which is zero).

Example:

```
int *ptr = NULL;
```

NULL pointers cannot be de-referenced.

for example if ptr is pointed to null then *p would cause an error.

DATE........................

QUESTION NO. 03

"PUBLIC, PROTECTED, AND PRIVATE INHERITANCE."

Inheritance Access chart:

		Inheritance		
		public	Protected	Private
Access	Public	public	Protected	Private
	Protected	Protected	Protected	private
	Private	Private	Private.	Private

Example #01 :-

```
        class A {
private: int priA;
protected: int proA;
public: int pubA;
            void displayA() {
            cout << priA << proA << pubA ;}
        };
```

```
    class B : public A {
private: int priB ;
protected: int proB ;
public: int pubB ;
        void displayB() {
        cout << priA << proA << pubA ;
        cout << priB << proB << pubB ;
        }
    };
  int main () {
    A objA;  B objB;
    cout << objA.priA << objA.proA
    << obj A.pubA << endl;
    cout << objB.priA << objB.proA
    << objB.pubB << endl;
    cout << objB.priB << objB.proB
```

Example #02 :-

```
    class A
private: int priA;
protected: int proA;
public: int pubA;
        void display () {
        cout << priA << proA << pubA ; }
    };
```

```
        class B : protected A {
private : int priA;
protected: int proA;
public: int pubA;
        void display() {
         cout <<priA <<proA <<pubA; }
        };
class B : protected A{
 private : int priB;
protected : int~~0~~ proB;
public : int pubB;
          void display() {
          cout<<priA <<proA <<pubA;
          cout<<priB <<proB <<pubB; }
          };
 int main() {
      A objA: BobjB;
      cout<<objA.priA <<objA.proA
      << obj.A.pubA<<endl;
       cout <<objB.priA <<objB.proA <<objB.pubA
       <<endl;
       cout <<objB.priB <<objB.proB.
```

Example #03:-

```
      class A {
private : int priA;
protected : int proA;
public : int pubA;
        void display() {
            cout<<priA<<proA<<pub A; }
        };
   class B : private A {
 private : int priB;
 protected : int proB;
 public : int pubB;
        void display() {
      cout<<priA<<proA<<pubA;
      cout<<priB<<proB<<pubB; }
       };

int main() {
     A objA; B objB;
     cout<<objA.priA<<objA.proA
      <<objA.pubA<<endl
      cout<<objB.priA<<objB.proA<<objB.pubA
       <<endl;
      cout<<objB'priB<<objB.proB.
```

DATE.......................

Example # 04 :-

```
   class A {
private: int priA;
protected: int proA;
public: int pubA;
       void display() {
cout << priA << proA << pubA;
   }
};
```

```
  class B: public A {
private: int priB;
protected: int proB;
public: int pubB;
       void display() {
cout << priA << proA << pubA;
cout << priB << proB << pubB;
 }
};
```

```
class c: public B {
private: int priC;
protected: int proC;
public; int pubC;
   void display(){
cout << priA << proA << pubA;
cout << priB << proB << pubB;
cout << pric << proc << pubc;
    }
};
```

```
int main(){
 A objA; BobjB; C objC;
  cout<<objA.priA<<objA.proA
   <<objA.pubA<<endl
  cout<<objB.priA<<objB.proA
   <<objB.pubA<<endl;
  cout<< objB.priB<<objBproB.
    <<objB.pubB <<endl;
  cout <<objC.priA<<objC.proA;
   <<objC. pubA <<endl;
  cout<<objC.priB<<objC. proB
     <<objC.pubB<<endl;
  cout<<objC.priC<<objC.proC
      <<objC.pubC <<endl;
```

Date________________

Example #05:-

```
class A {
private: int priA;
protected; int proA;
public; int pubA;
    void display(){
cout<<priA<<proA<<pubA;
   }
};
```

```
class B : protected A {
private: int priB;
protected: int proB;
public: int pubB;
      void display(){
cout<<priA<<proA<<pubA;
cout<<priB<<proB<<pubB;
  }
 };
```

```
class C : public B{
private:
protected:
public:
   void display(){
cout<<priA<<proA<<pubA;
cout<<priB<<proB<<pubB;
cout<<priC<<proC<<pubC;
   }
   };
```

Date

```
int main() {
A objA; B objB; C objC;
cout<<objA.priA <<objA.proA
    <<objA.pubA<<endl;
cout<<objB.priA<<objB.proA.
     << objB.pubA <<endl.
cout<<objB.priB <<objB.proB
      <<objB.pubB<<endl;
cout <<objC.priA <<objC.proA
      <<objC.pubA << endl;
cout<<objC.priB<<objC.proB
      <<objC.pubB<<endl;
cout<< objC.priC<<objC.proC
      <<objC.pubC<<endl;
```

Example #06.

```
class A {
private: int priA;
protected: int proA;
public: int pubA;
  void display(){
cout<<priA<<proA<<pubA;
     }
};
```

```
class B: private A {
private: int priB;
protected: int proB;
public: int pubB;
  void display(){
cout<<priA<<proA<<pubA;
cout<<priB<<proB<<pubB;
     }
};
```

```
class C: public B {
private: int pric;
protected: int proc;
public: int pubC;
  void display(){
cout<<priA<<proA<<pubA;
cout<<priB<<proB<<pubB;
cout<<pric<<proC<<pubC;
    }
};
```

Date

```
int main() {
 A objA; B objB; C objC;
cout<<objA.priA <<objA.proA
    <<objA.pubA<<endl;
cout<<objB.priA<< objB.proA
     <<objB.pubA<<endl;
cout<<objB.priB << objB.proB
      <<objB.pubB <<endl;
cout<<objC.priA<<objC.proA
      <<objC.pubA<<endl;
cout<<objC.priB<<objC.proB
     <<objC.pubB <<endl;
cout<<objC.priC <<objC.priC
     <<objC.pubC <<endl;
```

DATE..........................

QUESTION NO. 04
"POLYMORPHISM"

Early & Late Binding
Virtual Functions.

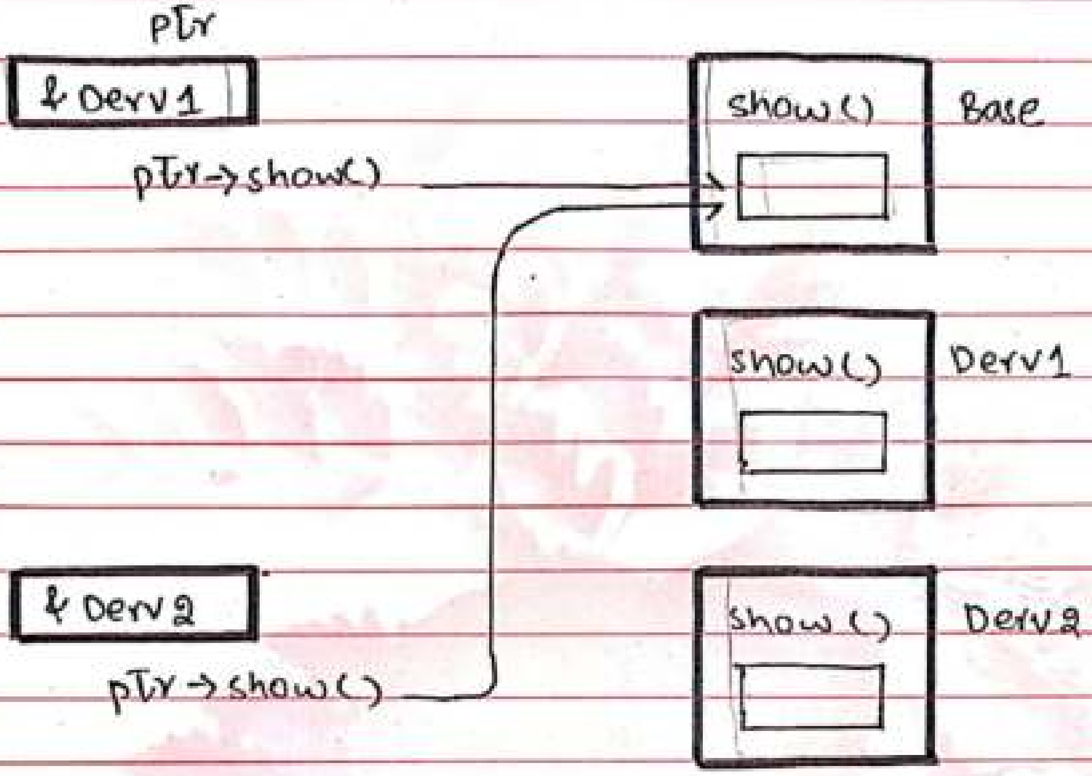
```
class base//base class.              {Early
 {                                     Binding}
public:
  void show() //normal function.
      { cout << "Base\n"; }
   };
  class Derv 1: public Base //derived.
   class 1 {
   public:
 void show() {
     cout << "Derv1 \n"; }
  };
 class Derv2: public Base // derived
 class 2 {
 public:
     void show() {
 cout<<" Derv2 \n"; }
};
```

```
  int main() {
     Base b;
    Derv1dv1;
 //object of derived class 1:
     Derv2dv2;
 // object of derived class 2
      Base* ptr;
 //pointer to base class
       ptr= &b;
       prt->show();
    ptr= &dv1;
//put address of dv1 in pointer
      ptr->show();
 //execute show()
      ptr = &dv2;
// put address of dv2 in pointer
      ptr->show();
 // execute show()
      return 0;
 }.
```

OUTPUT

```
Base
Base
Base
```

Early Binding:



```
int main() {
    Base b;
   Derv1 dv1; //object of derived class 1.
   Der2dv2; //object of derived class 2.
   Base* ptr; //pointer to base class.
   ptr = &b;
   ptr->show();
   ptr= &dv1; //put address of dv1 in pointer.
    ptr-> show(); // execute show()
    ptr = &dv2; //put address of dv2 in pointer.
    ptr->show(); //execute show().
```

DATE.......................

```
 return 0;
 }.
```

```
OUTPUT
Base
Base
Base
```

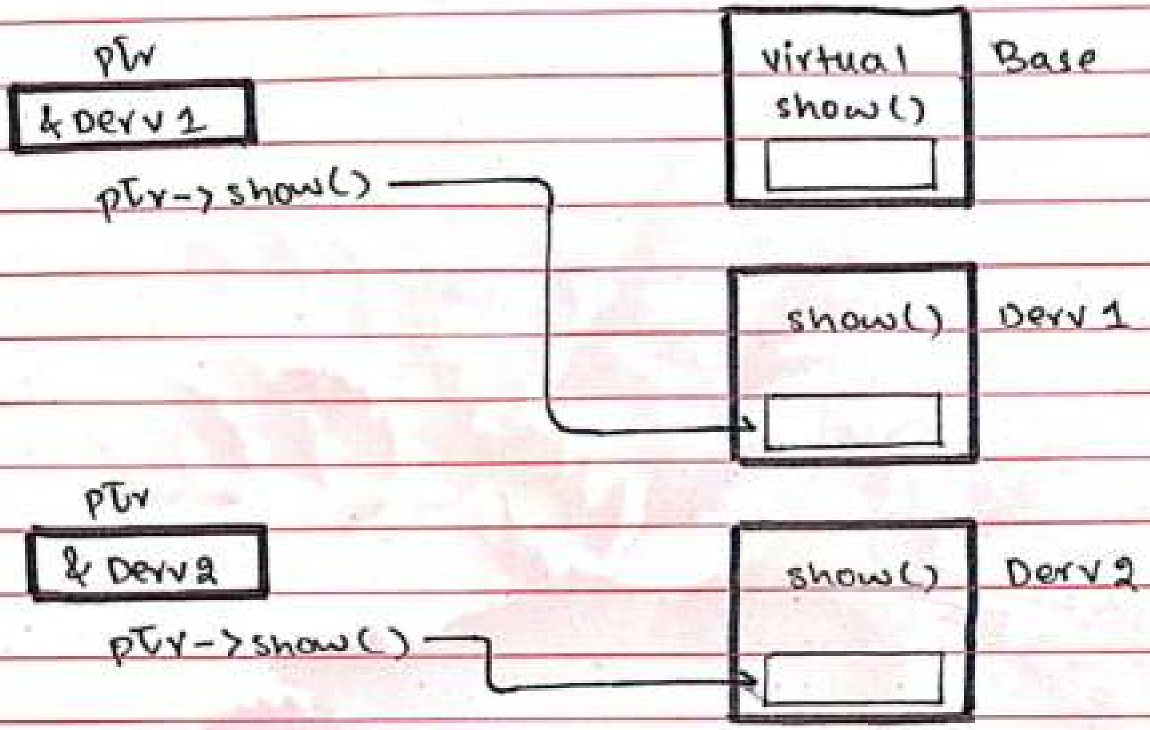
Late Binding
Virtual Member Functions

```
class Base // base class {
 public:  →{keyword}
  virtual void show() // virtual function.
 { cout << "Base\n"; }.
 };
class Derv1 : public Base //derived class 1.
{ public :
      void show () {
  cout << "Derv1\n"; }.
  };
class Derv2: public Base //derived class 2.
 { public:
      void show () {
  cout << "Derv2\n" ; }
 };
 int main() {
      Base b;
      Derv1dv1;
 //object of derived class 1
       Derv2dv2;
 //object of derived class 2
       Base *ptr;
 //pointer of base class.
       ptr = &b;
       prt-> show();
       ptr= &dv1;
 //put address of dv1 in pointer
      ptr-> show();
 //execute show()
      ptr= &dv2;
 //put address of dv2 in pointer
      ptr->show();
 //execute show()
       return 0;
 }
```

OUTPUT

```
Base
Derv1
Derv2
```

Late Binding:



```
int main () {
    Base b;
  Derv1dv1; // object of derived class 1.
  Derv2dv2;  //object of derived class 2.
   Base *ptr; //pointer to base class.
    ptr= &b;
    prt->show();
    ptr = &dv1;  // put address of dv1 in pointer.
     ptr->show(); //execute show ()
     ptr= &dv2; //put address of dv2 in pointer.
     ptr->show(); //execute show()
     return 0;
};
```

OUTPUT

```
Base
Derv1
Derv2.
```

Abstract classes and Pure Virtual functions

```
class Base // base class
{ public:
      virtual void show () =0; // pure virtual function
};
class Derv1: public Base // derived class 1
 { public:
      Void show ()
  { cout << "Derv1 \n"; }
};
class Derv2: public Base // derived class 2.
{ public:
      void show ()
    { cout << "Derv2 \n"; }.
};
```

DATE.....................

```
  int main(){
// Base bad;
// cant' make object from
    abstract class
       Base *arr[2];
//array of pointers to base class
        Derv1dv1;
//object of derivedclass 1.
        Derv2dv2;
//object of derived class 2.
        arr[0]= &dv1.
//put address of dv1 in array.
        arr[1] = &dv2
//put address of dv2 in array
        arr[0]->show();
//.execute show() in both obj
         arr[1]-> show();
          return 0;
 }
```

OUTPUT

```
Derv1
Derv2
```