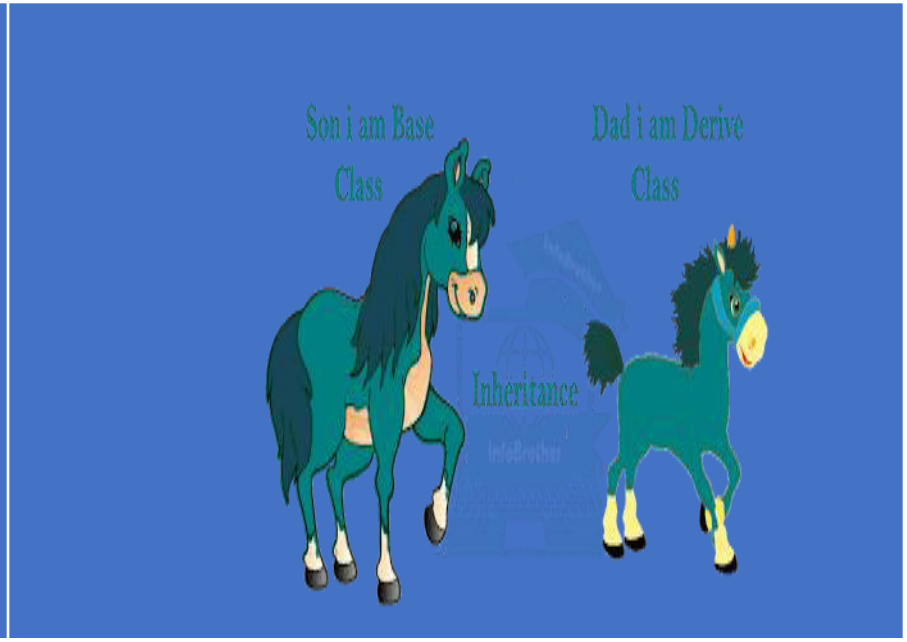
# Unit Contents

**Inheritance-** Base Class and derived Class, protected members, relationship between base Class and derived Class, Constructor and destructor in Derived Class, Overriding Member Functions, Class Hierarchies, Public and Private Inheritance, Types of Inheritance, Ambiguity in Multiple Inheritance, Virtual Base Class, Abstract class, Friend Class , Nested Class.

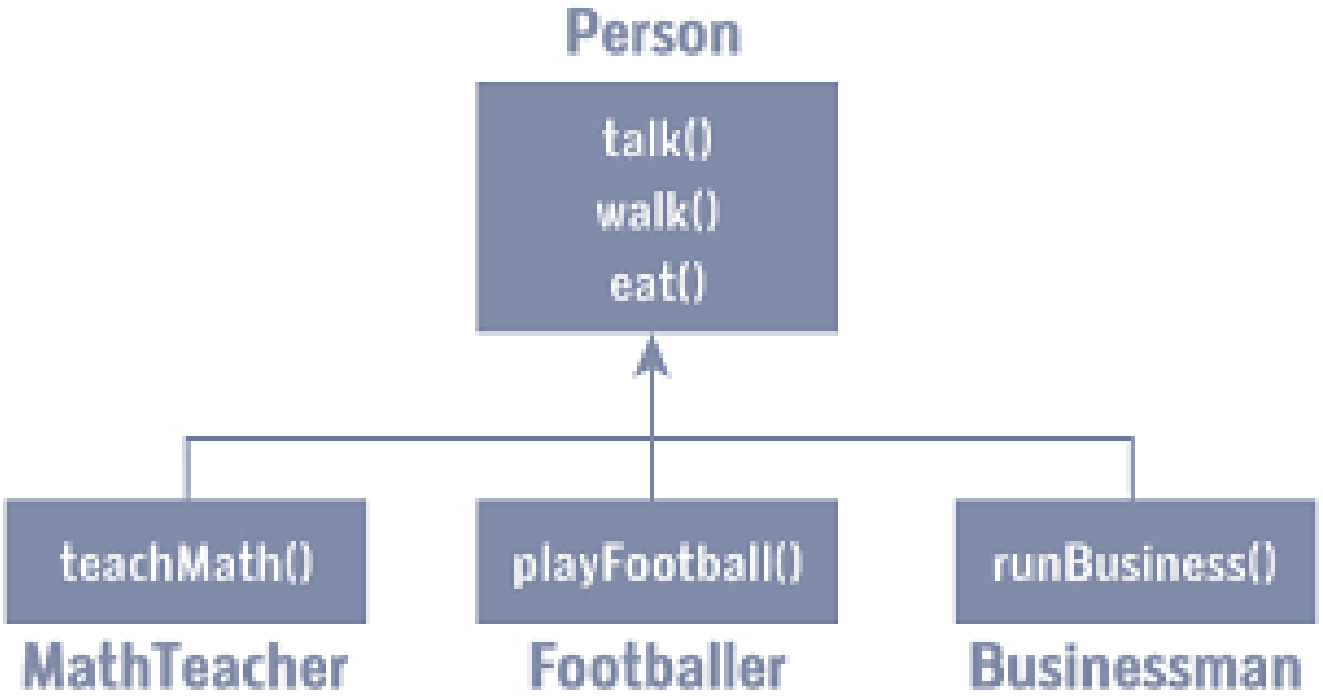
**Pointers**: declaring and initializing pointers, indirection Operators, Memory Management: new and delete, Pointers to Objects, this pointer, Pointers Vs Arrays, accessing Arrays using pointers, Arrays of Pointers, Function pointers, Pointers to Pointers, Pointers to Derived classes, Passing pointers to functions, Return pointers from functions, Null pointer, void pointer.

# Inheritance

## What is Inheritance?

# Inheritance



# Basic Syntax

```
Class BaseClass
{
        //BaseClass properties and functions
};

class DerivedClass : accessSpecifier BaseClass
{

};
```

Access specifier can be public, protected and private.
The default access specifier is **private.**

# Access Specifiers

- Public - members are accessible from outside the class
- Private - members cannot be accessed (or viewed) from outside the class
- Protected - members cannot be accessed from outside the class, however, they can be accessed from inherited classes.

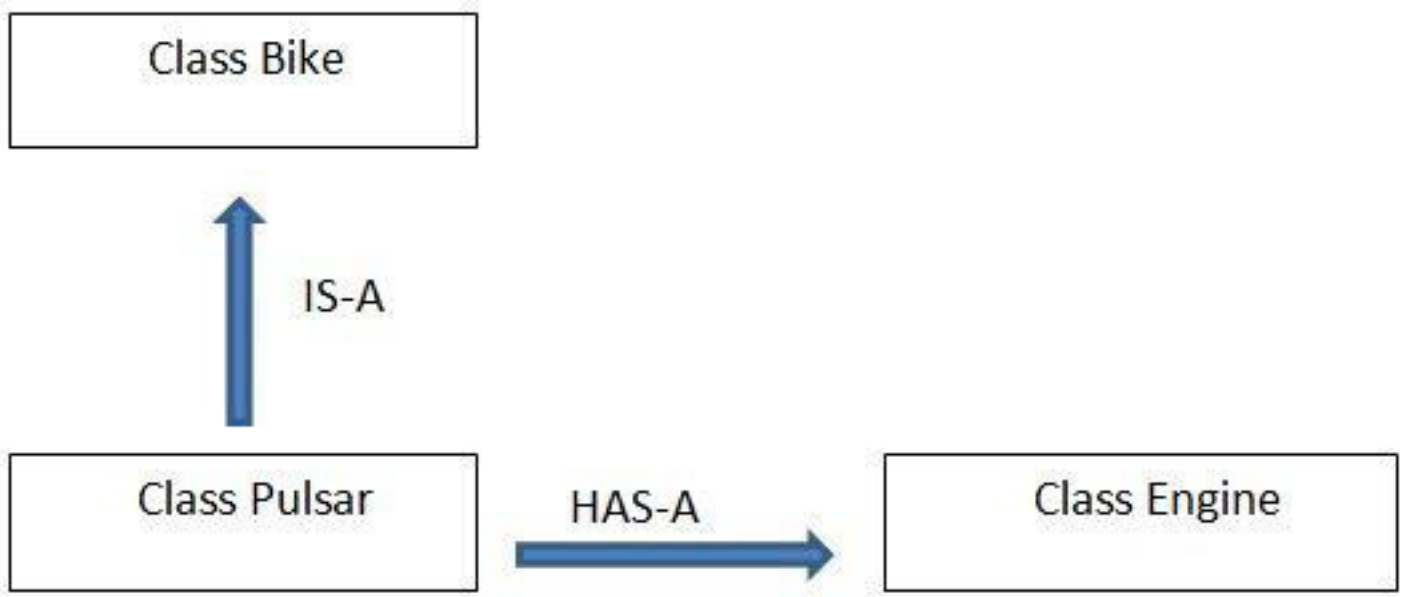
# Access Specifiers

| Access Specifier | Within same class | In derived class | Outside the class |
|---|---|---|---|
| Private | Yes | No | No |
| Protected | Yes | Yes | No |
| Public | Yes | Yes | Yes |

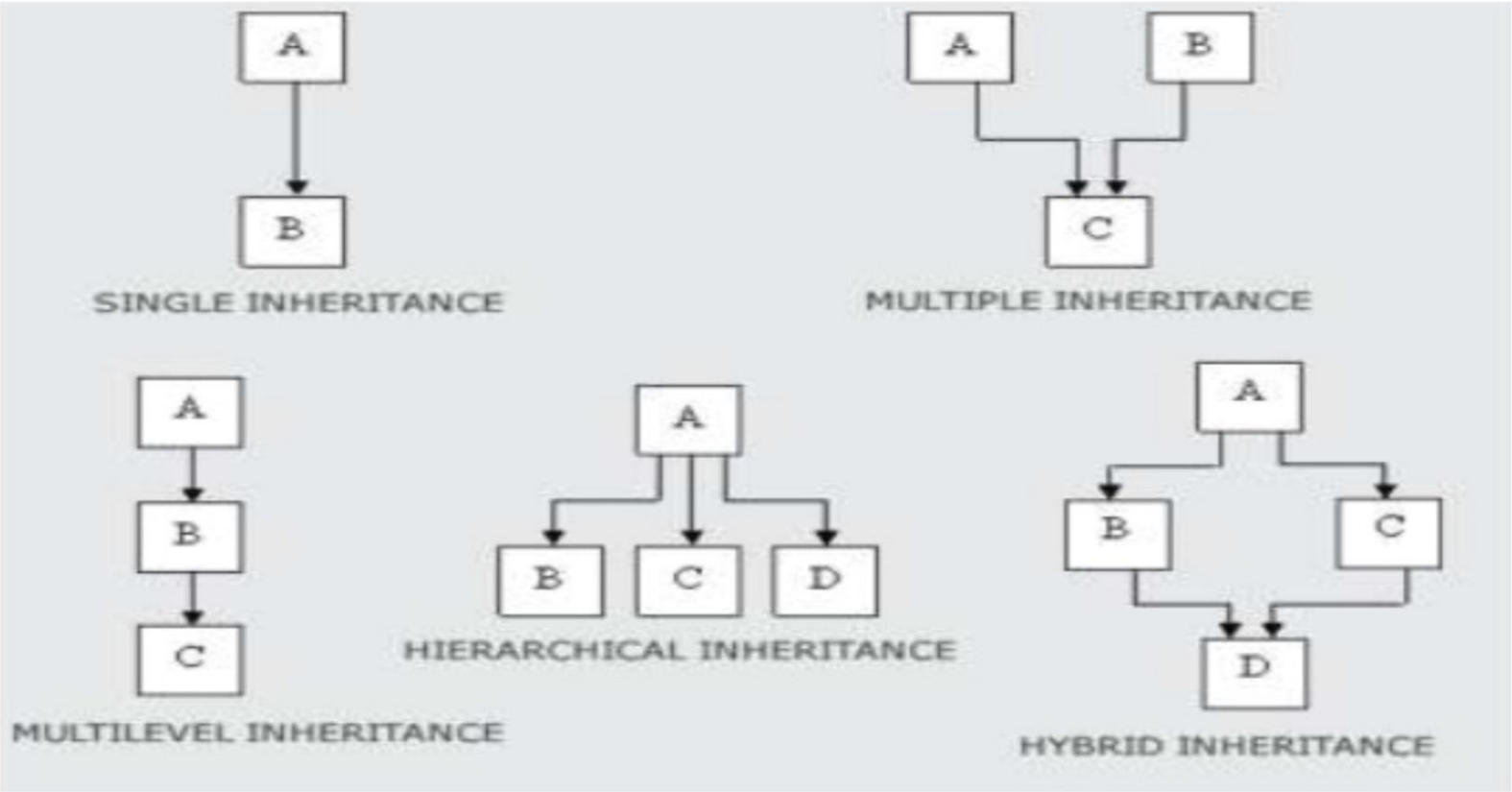
# Relationships in OOP

- One of the advantages of an Object-Oriented programming language is code reuse.
- There are two ways we can do code reuse either by the implementation of inheritance (IS-A relationship), or object composition (HAS-A relationship).
- IS- A Relationship : based on Inheritance
- HAS-A Relationship : the use of instance variables that are references to other objects. HAS-A relationship is composition.

# Relationships in OOP



# Types of Inheritance



# Ambiguity in Multiple Inheritance

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

```
class derived : public base1, public base2
{
};
int main()
{
            derived obj;
            obj.someFunction()          // Error!
            return 0;
}
```

# Ambiguity in Multiple Inheritance…

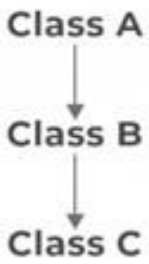
- This problem can be solved using scope resolution operator. e.g.

```
int main()
{
        obj.base1::someFunction( );      // Function of base1 class is called
        obj.base2::someFunction();       // Function of base2 class is called
}
```

# Constructor and destructor in Derived Class

- A derived-class constructor :
  - Calls the constructor for its base class first to initialize its base-class members.
  - If the derived-class constructor is omitted, its default constructor calls the base-class' default constructor
- A derived-class destructor :
  - Destructors are called in the reverse order of constructor calls :
  - So a derived-class destructor is called before its base class destructor

# Order of calling Constructors and Destructors in Inheritance




**Order of Constructor Call**

A( ) - Class A Constructor

B( ) - Class B Constructor

A( ) - Class C Constructor

**Order of Destructor Call**

C( ) - Class C Destructor

B( ) - Class B Destructor

A( ) - Class A Destructor

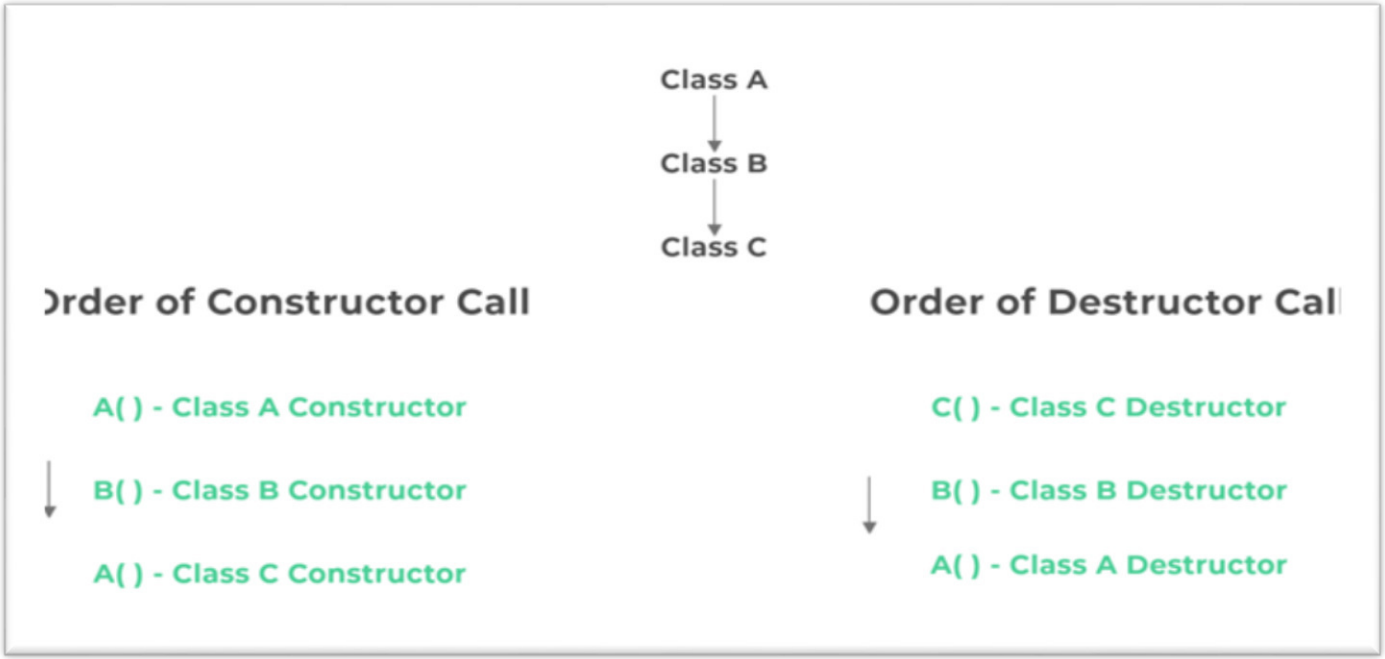
# Guess output ??

```
class parent                    //parent class
{
        public:
        parent() //constructor
        {
                cout<<"Parent Constructor\n";
        }

        ~parent()               //destructor
        {
                cout<<"Parent Destructor\n";
        }
};
```

```
class child : public parent             //child class
{
        public:
        child() //constructor
        {
                cout<<"Child Constructor\n";
        }

        ~ child() //destructor
        {
                cout<<"Child Destructor\n";
        }
};
int main()
{
        child c;
        return 0;
}
```

# Program Output

Parent Constructor
Child Constructor
Child Destructor
Parent Destructor

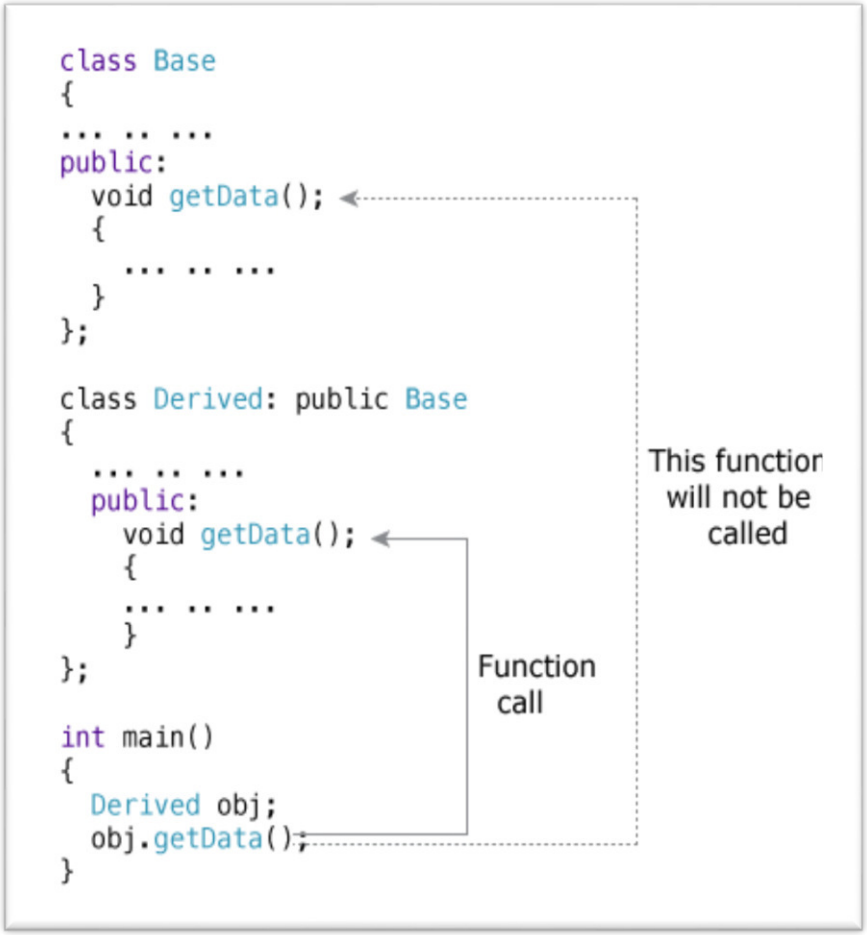


# Overriding Member Functions

## Requirements for Overriding a Function :

- Inheritance should be there.
  - Function overriding cannot be done within a class. For this we require a derived class and a base class.
- Function that is redefined must have exactly same signature in both base and derived class, that means same name, same return type and same list of parameters.

# Overriding Member Functions




- Derived class object will invoke the derived class function.
- How to access the overridden function of the base class??

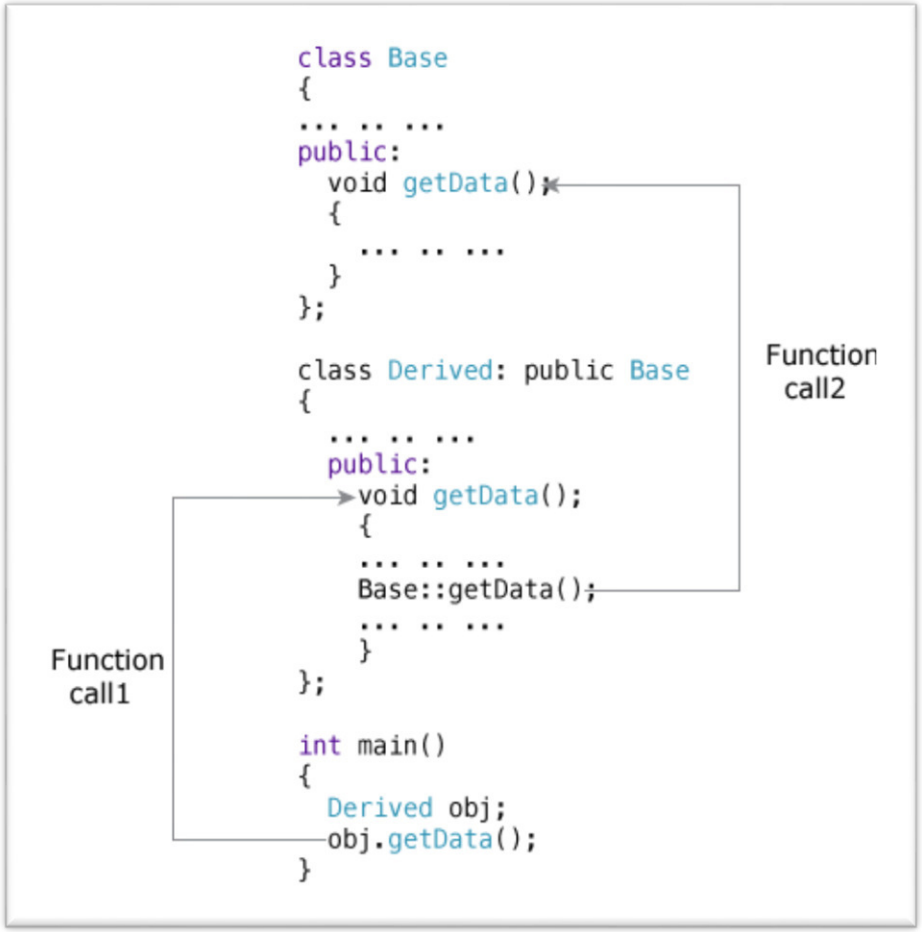
# First Solution

```
class base
{
  public:
  void getdata()
  {
     cout<<"base..";
  }
};
class derived:public base
{
  public:
  void getdata()
  {
     cout<<"derived..";
  }
};
```

```
int main()
{
   derived obj;
   obj.base::getdata();
   obj. getdata();
   return 0;
}
```

Invoke the base class function using base class name and scope resolution operator.

# Second Solution




Invoke the base class function from the derived class function using base class name and scope resolution operator.

# Public, Private and Protected Inheritance

| Base Class Specifer | public inheritance | protected inheritance | private inheritance |
|---|---|---|---|
| public | public in derived class | protected in derived class | private in derived class |
| protected | protected in derived class | protected in derived class | private in derived class |
| private | hidden | hidden | hidden |

# Virtual Base Class

- When two or more objects are derived from a common base class, we can prevent multiple copies of the base class being present in an object derived from those objects by declaring the base class as virtual when it is being inherited.
- Such a base class is known as virtual base class. This can be achieved by preceding the base class' name with the word virtual.

# Virtual Base Class

- The virtual base class is used when a derived class has multiple copies of the base class.

```
class B {
  public: int b;
};

class D1 : public B {
  public: int d1;
};

class D2 : public B {
  public: int d2;
};
```

```
class D3 : public D1, public D2 {
  public: int d3;
};
int main() {
  D3 obj;

  obj.b = 40; /*error will occur as multiple
copies of variable b are present in class D3*/

}
```

# Virtual Base Class

- The virtual base class is used when a derived class has multiple copies of the base class.

```
class B
{
   public: int b;
};
class D1 : virtual public B
{
   public: int d1;
};
class D2 : virtual public B
{
   public: int d2;
};
```

```
class D3 : public D1, public D2
{
   public: int d3;
};
int main()
{
   D3 obj;
   obj.b = 40;          //No error
}
```

# Friend Class

## Who is a Friend?

A friend is one who has access to all your "PRIVATE" stuff

# 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**.

  Syntax :

  ```
  friend class class_name;
  ```

# Friend Class Program

```
class A
{
        private:
                int a;
        public:
        A() { a = 0; }
   friend class B;
};
```

```
class B
{
        private:
                int b;
        public:
        void showA (A& x)
        {    // Since B is friend of A, it can access
             // private members of A
             cout << "A::a=" << x.a;
        }
};
int main()
{
   A a;    B b;
   b.showA(a);
   return 0;
}
```

# Nested Class

- A nested class is a class which is declared in another enclosing class.

e.g.

```
class Outer
{
    class  Inner
    {
    };
};
```

# Nested Class

```
class A
{
  public:
        class B
        {
              private:
              int num;
              public:
              void getdata(int n)
              {
                    num = n;
              }
        };
};
```

```
int main()
{
  cout<<"Nested classes in C++";
  A :: B obj;
  obj.getdata(9);
  return 0;
}
```

# Pointers

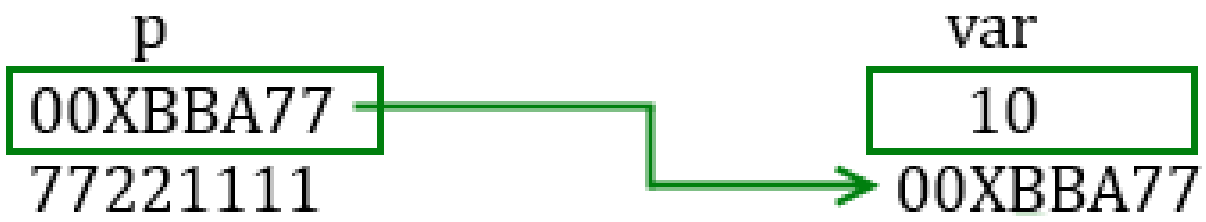
- Normal variable is used to store the value.
- A pointer is a variable that holds the address of another variable.
- Pointers are symbolic representations of addresses.
- We can have pointer to any variable type.

Syntax of Pointer :

```
data_type *pointer_name;
```

# Pointers

```
int var =10;
int *p;
p = &var;
```




P is an pointer here which is pointing to the address of variable var.

Note: Data type for var and p should be the same.

# Pointer Declaration

```
int *ip;          // pointer to an integer

double *dp;       // pointer to a double

 float *fp;       // pointer to a float

char *ch;         // pointer to character
```

# Reference operator (&) and Deference operator (*)

- Reference operator (&) gives the address of a variable.
- To get the value stored in the memory address, we use the dereference operator (*) which is also called **as indirection operator.**

e.g. If a number variable is stored in the memory address **0x123**, and it contains a value **5**.

- The **reference (&)** operator gives the value **0x123**, while the **dereference (*)** operator gives the value **5.**

# Reference operator (&) and Deference operator (*)

```
#include <iostream>
using namespace std;
int main()
{
        int a=5;
        cout<<a<<endl;
        cout<<&a<<endl;
        cout<<*&a;

        return 0;
}
```

```
5
0027FEA0
5
```

# Pointer Program

```
int main ()
 {
int var = 20;
int *ip;                    // pointer variable
ip = &var;      // store address of var in pointer

cout << "Value of var variable: "var << endl;
cout << "Address stored in ip variable: “<<ip;
cout << "Value of *ip variable: "<< *ip << endl;
return 0;
 }
```

**Output :**

Value of var variable: 20

Address stored in ip variable: 0xbfc601ac

Value of *ip variable: 20

# Memory Management

- Two operators **new** and **delete** that perform the task of allocating and freeing the memory in a better and easier way.

# The new operator

- syntax of new operator:

  pointer_variable = new datatype;

- syntax to initialize the memory :

  pointer_variable = new datatype(value);

- syntax to allocate a block of memory,

  pointer_variable = new datatype[size];

# The delete operator

- syntax of delete operator :

  delete pointer_variable;

- syntax to delete the block of allocated memory:

  delete[ ] pointer_variable;

# Program of new and delete operator

```
int main ()
{
    int *ptr1 = NULL;
    ptr1 = new int;
    *ptr1 = 28;

    float *ptr2 = new float(299.121);

    int *ptr3 = new int[5];

    cout << "Value of pointer var1 : " << *ptr1;
    cout << "Value of pointer var2 : " << *ptr2;

    if (!ptr3)
    {
            cout << "Allocation of memory failed\n";
    }
```

```
else
{
    for (int i = 0; i < 5; i++)
            ptr3[i] = i+1;

    cout << "Value stored in block of memory:";

    for (int i = 0; i < 5; i++)
            cout << ptr3[i] << " ";
  }
 delete ptr1;    delete ptr2;    delete[] ptr3;
 return 0;
 }
```

**Output**

```
Value of pointer var1 : 28
Value of pointer var2 : 299.121
Value stored in block of memory: 1 2 3 4 5
```

# Pointers to Objects

- Just like other pointers, pointers to objects are declared by placing * in front of an object pointer's name.

- Syntax:

  Class_name *object_pointer ;

# Pointers to Objects

```
class myclass
{
        int i;
  public:
        void read(int j)
        {
                i= j;
        }

        int getint()
        {
              return i;
        }
};
```

```
void main()
{
        myclass ob, *objectPointer;

        //initialize pointer
        objectPointer = &ob;

        objectPointer->read(10);

        cout<<objectPointer->getint();
}
```

Output:
10

# this pointer

- It holds the address of current object, in simple words you can say that it points to the current object of the class.
- It can be used to pass current object as a parameter to another method.
- It can be used to refer current class instance variable.

# this pointer

```
class Demo
{
private:
        int num;
        char ch;
public:
        void setValues(int num, char ch)
        {
                this->num =num;
                this->ch=ch;
        }
        void displayValues()
        {
                 cout<<num<<endl;
                cout<<ch;
        }
};
```

```
int main()
{
        Demo obj;

        obj.setValues(100, 'A');
        obj.displayValues();

        return 0;
}
```

Output :

100

A

# Pointers Vs Arrays

| Declaration<br>//In C++<br>type var_name[size]; | Declaration<br>//In C++<br>type * var_name; |
|---|---|
| Stores the value of the variable of homogeneous data type. | Store the address of the another variable of same datatype. |
| An array of pointers can be generated. | A pointer to an array can be generated. |
| An array can store the number of elements, mentioned in the size of array variable. | A pointer variable can store the address of only one variable at a time. |
| Used to allocate fixed sized memory. | Used for dynamic memory allocation. |

# Accessing Array Elements Using Pointer

```
#include <iostream>
using namespace std;
int main()
{
    int arr[5] = {5, 2, 9, 4, 1};
    int *ptr = &arr[2];
    cout<<"The value in the second index of the array is: "<< *ptr;
   return 0;
 }
```

Output :

The value in the second index of the array is: 9

# Array and Pointer

```
int main()
{
        int a[4], i;
        int*ptr;
        for(i=0;i<5;i++)                    // Elements inserted into an array
        {
                cin>>a[i];
        }
        ptr=a;                              //pointer initialized with base address of an array
        for(i=0;i<5;i++)
        {
                cout<<*(ptr+i);             //array is printed using the pointer
        }
        return 0;
}
```

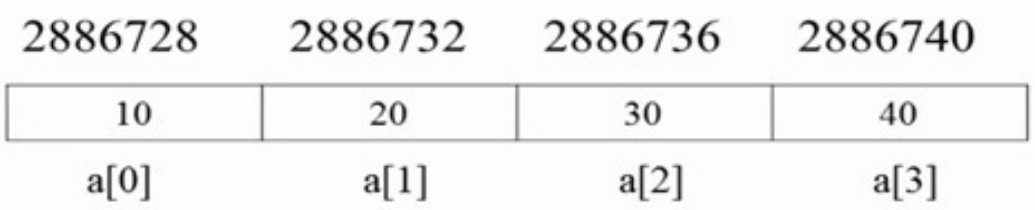
# Observations: Array and Pointer

```
int array[4];
int *ptr = array; // Same as &array[0]

// All the statements below are
// semantically identical.
//
array[2] = 7;      // Normal array access.
ptr[2]   = 7;      // Treat pointer like an array.

*(ptr + 2)   = 7; // Use pointer arithmetic.
*(array + 2) = 7; // Array name yields its address.
```

# Arrays of Pointers

- Arrays of Pointers : Addresses of array elements.

```
int a [ ] = {10,20,30,40};

int *p[4];      // *p[0], *p[1], *p[2], *p[3]
```



# Array of Pointers

```
#include <iostream>
using namespace std;
const int MAX = 4;
int main ()
{
 int a[MAX] = {10, 20, 30,40};
 int *p [MAX];
 for (int i = 0; i < MAX; i++)
{
   // assign the address of integers
   p [i] = &a[i];
 }
```

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

Output :

```
Value of var[0] = 10
Value of var[1] = 20
Value of var[2] = 30
Value of var[3] = 40
```

# Function Pointers

- Pointer contains the address of function.
- Function name is starting address of code that defines function.

**Syntax :**

**Ret_type (fun_pointer)(arg.);**

**e.g.**

```
int (*ftr)(int,int);
```

**Function pointers can be**

- Passed to functions
- Returned from functions.
- Stored in arrays
- Assigned to other function pointers

# Program of Function Pointers

```
void one(int a, int b)
{
   cout << a+b << "\n";
}

void two(int a, int b)
{
   cout << a+b << "\n";
}
```

```
int main()
{
        //Declare a function pointer
        void (*fptr)(int, int);

        fptr = one;
        fptr(12, 3);        //=> one(12, 3)

        fptr = two;
        fptr(12, 2);        //=> two(12, 2)

        return 0;
 }
```

Output : 15
10

# Function Pointer to class member function

```
#include <iostream>
using namespace std;
class Data
{
   public:
   int f(float)
   {
      return 10;
   }
};
```

```
int main()
{
   int (Data::*fp2) (float);     // Declaration
   Data obj;

   fp2 = &Data::f;   // Assignment

   cout<<(obj.*fp2)(20.0);
}
```

# Function Pointer to class member function

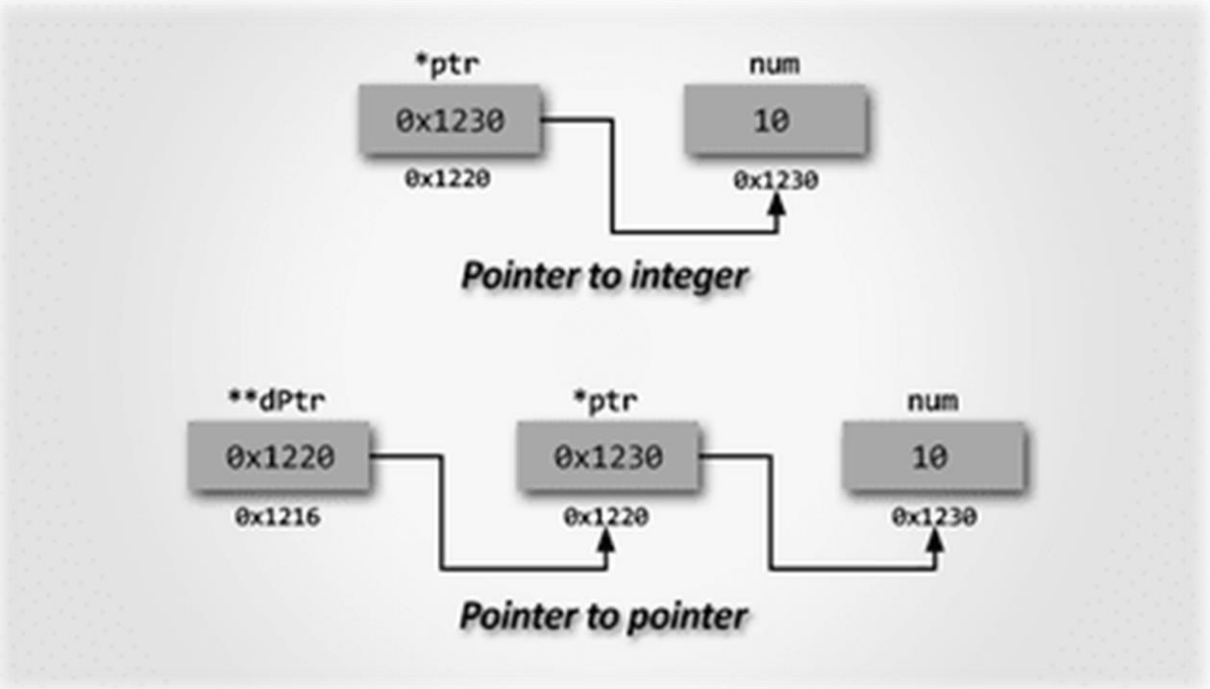
```
class sample
 {
public:
 int i;
 Number()
    { i = 0; }
int one()
   { return i+1; }
int two()
   { return i+2; }
};
```

```
int main()
{
 sample object;
 int (sample::*NumberPtr)();

 NumberPtr = & sample ::one;
 cout << (object.*NumberPtr)() << endl;

 NumberPtr = & sample ::two;
 cout << (object.*NumberPtr)() << endl;

 return 0;
}
```

**Output : 1  2**

# Pointers to Pointers

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




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.

# Pointers to Pointers

```
int main()
{
  int *vptr;
  int ** intptr;
  int var = 10;

  vptr = &var;
  intptr = &vptr;

  cout<<"Variable var: "<<var<<endl;
  cout<<"Pointer to Var: "<<*vptr<<endl;
  cout<<"Pointer to Pointer : "<<**intptr;
  return 0;
}
```

**Output :**

Variable var: 10
Pointer to Variable: 10
Pointer to Pointer to a variable: 10

# Pointers to Derived classes

- Pointers can be declared to point base or derived classes.
- Base class pointer can point to objects of base and derived class.
- Pointer to derived class object cannot point to objects of base class.

# Pointers to derived class

```
class Base
{
   public:
      int x;
};

class derived:public Base
{
  public:
      int y;
};
```

```
int main()
{
      Base *base_ptr;
      derived der_ob;
      base_ptr=&der_ob;

      derived*der_ptr;
      Base base_ob;
      der_ptr=&base_ob;

      return 0;
}
```

Will work

Won't work

# Pointers to derived class

```
#include <iostream.h>
class Base
{
        public:
        int x;
        void show ()
        {
                cout<<"X="<<x<<endl;
        }
};
class Derive: public Base
{
        public:
        int y;
        void display ();
        {
                cout<<"X="<<x<<endl;
                cout<<"Y="<<y<<endl;
        }
};
```

```
int main ()
{
        Base B1;
        Base *ptr;
        ptr = &B1;
        ptr->x = 10;
        ptr->show();

        Derive D1;
        Derive *ptr1;
        ptr1 = &D1;
        ptr1->x = 10;
        ptr1->y = 20;
        ptr1->display ();
}
```

- Output:
  X= 10
  X = 10
  Y = 20

# Access derived class member from base class pointer : using typecast

```
#include <iostream.h>
class Base
{
        public:
        int x;
        void show ()
        {
                cout<<"X="<<x<<endl;
        }
};
class Derive: public Base
{
        public:
        int y;
        void display ();
        {
                cout<<"X="<<x<<endl;
                cout<<"Y="<<y<<endl;
        }
};
```

```
int main()
{
 Derive D1;
 Base *ptr;
 ptr = &D1;
 ptr->x = 10;
 ptr->show();

 static_cast<Derive*>(ptr)->y=20;
 static_cast<Derive*>(ptr)->display ();
return 0;
}
```

# Passing Pointers To Functions

- C++ allows you to pass a pointer to a function. To do so, simply declare the function parameter as a pointer type.
- Advantages :

- Flexible: can change the value of the object.

- Fast: only need to copy a pointer not the whole object.

- Disadvantages:

- Unsafe: can change value of the variable in calling function.

- Complex: variable is pointer so needs * or -> to access.

# Passing Pointers To Functions

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

void swap(int* a, int* b)
{
   int temp;
   temp = *a;
   *a = *b;
   *b = temp;
}
```

```
int main()
{
   int a, b;
   cout<<"Enter values for swapping";
   cin>>a>>b;

   swap(&a,&b);

  cout<<"Swapped values"<<endl;
  cout<<"a = "<<a<<"\t"<<"b = "<<b;
  return 0;
}
```

**Output:**

Enter values for swapping:  3 2
Swapped values
a = 2 b = 3

# Return Pointers from Functions

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

int* show (int*m)
{
   *m=*m*10;
     return m;
}
```

```
int main()
{
   int *x;
   int y=10;

   x=show(&y);

   cout<<"\nOutput is ="<<*x;
   return 0;
}
```

Output:
Output is 100

# Return Pointers from Functions

- Pointers is a variable which is used to store the memory address of another variable.
- It is not recommended to return the address of a local variable outside the function as it goes out of scope after function returns.
- So to execute the concept of returning a pointer from function in C/C++ you must define the local variable as a static variable.

# Return Pointers from Functions

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

int* show ()
{
  static int p;

   cout<<"Enter value";
   cin>>p;

   return &p;
}
```

```
int main()
{
   int *x;
   x=show();

   cout<<"\nValue=";
   cout<<*x;

   return 0;
}
```

# Null pointer

- Besides memory addresses, there is one additional value that a pointer can hold: a null value.
- A **null value** is a special value that means the pointer is not pointing at anything.
- A pointer holding a null value is called a **null pointer**.

```
#include <iostream>
 using namespace std;
 int main ()
{
   int *ptr = NULL;
   cout << “ptr value = " << ptr ;
   return 0;
}
```

**Output :**

ptr value = 0

# Void pointer

- It holds the address of any data type, but it is not associated with any data type.
- Syntax :

```
void *ptr;
```

- The size of void pointer varies system to system. For 16 bit system it is 16-bit. For 32 bit system, it is 32-bit and for 64 bit system the size is 64-bit.

- In C++, you cannot assign the address of variable of one type to a pointer of another type.

  e.g.

```
int *ptr;
double d = 9;
ptr = &d; // Error: can't assign double* to int*
```

- To avoid it, make use of general purpose pointer i.e. void pointer.

# Void pointer

```
#include <iostream>
using namespace std;
 int main()
 {
   void* ptr;
   float f = 2.3;

   ptr = &f;

   cout << &f << endl;
   cout<<ptr;
   cout<<*((float*)ptr);

   return 0;
 }
```

**Output :**

0xffd117ac
0xffd117ac
2.3

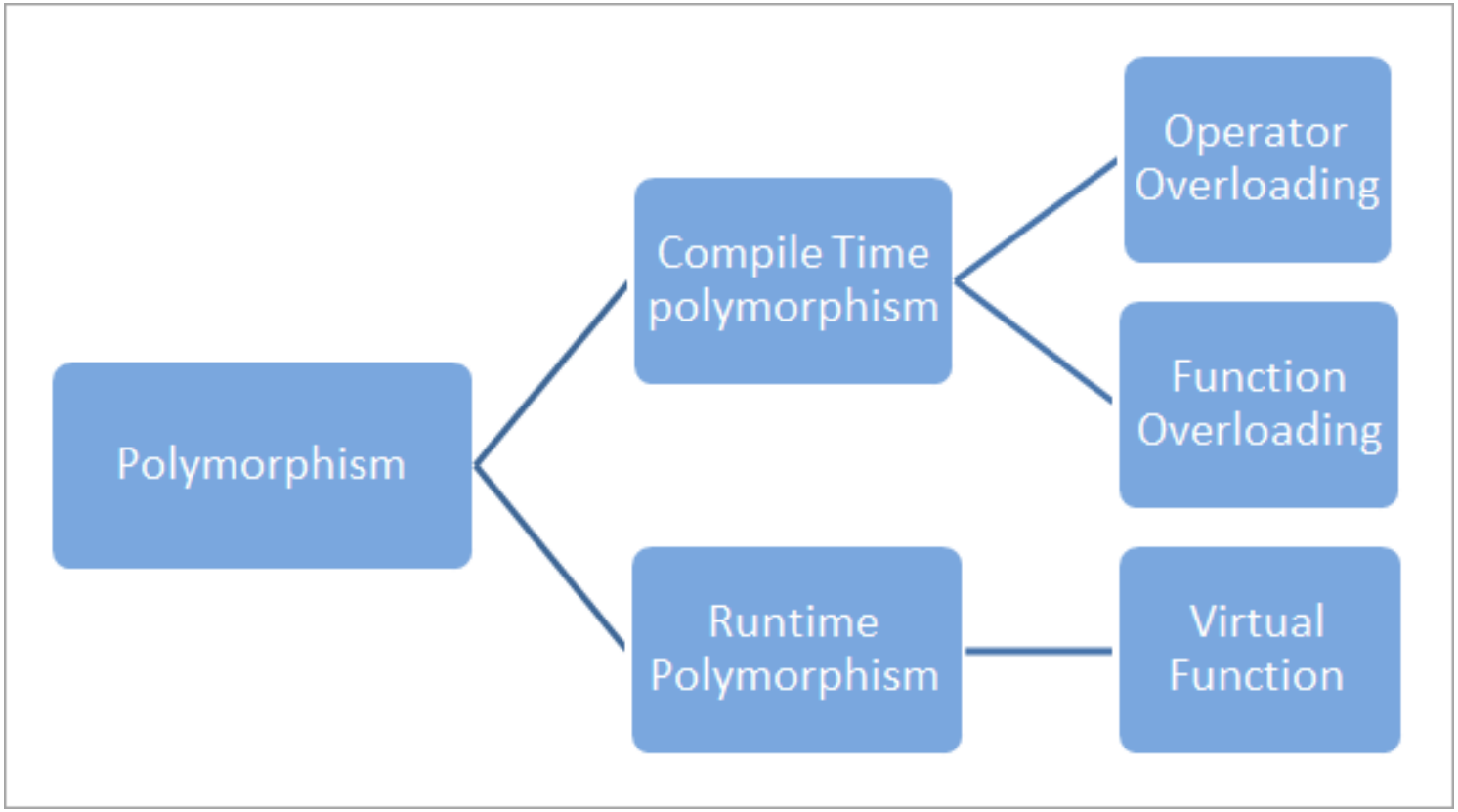
# Case Study

## Firefox developed using C++:

Refer the following link.

+https://www-archive.mozilla.org/hacking/coding-introduction

# Virtual Function



# Rules for Virtual Function

- Must be members of some class.
- Cannot be static member.
- Accessed through object pointers.
- Can be a friend of another class.
- Must be defined in the base class, even though it is not used.
- The prototypes of a virtual function of the base class and all the derived classes must be identical.

# Virtual function

```
class A
{
        public:
        virtual void display()
        {
        cout << "Base class is invoked"<<endl;
        }
};
class B:public A
{
        public:
        void display()
        {
        cout << "Derived Class is invoked"<<endl;

         }
};
```

```
int main()
{
        A* a,ob;    //pointer of base class
        B b;     //object of derived class

        //call derived class function
        a = &b;
        a-
>display();   //Late Binding occurs

        //call base class function
        a = &ob;
        a-
>display();   //Late Binding occurs
}
```

Output :

Derived Class is invoked

Base class is invoked

# Pure Virtual Function

- declared in the base class that has no definition relative to the base class.
- Syntax :

  virtual return type fun_name=0;

- e.g.

  virtual void show()=0;

# Virtual function program

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

class Base
{
   int x;
public:
    virtual void fun() = 0;
    int getX() { return x; }
};

// This class inherits from Base and implements fun()
class Derived: public Base
{
    int y;
public:
    void fun() { cout << "fun() called"; }
};

int main(void)
{
    Derived d;
    d.fun();
    return 0;
}
```

- A class is abstract if it has at least one pure virtual function.
- If we do not override the pure virtual function in derived class, then derived class also becomes abstract class.
- Abstract class can't be instantiated directly.
- An abstract class can have constructors.
- We can have pointers and references of abstract class type.