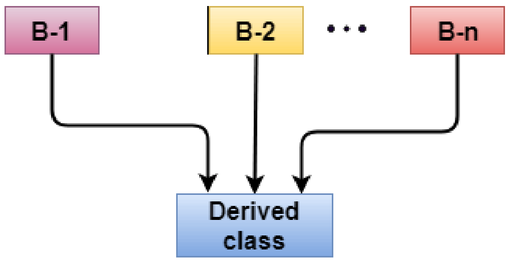
Module 4

Inheritance, Pointers, Virtual Functions, Polymorphism

- Inheritance in Object Oriented Programming can be described as a process of creating new classes from existing classes. ... An existing class that is "parent" of a new class is called a base class. New class that inherits properties of the base class is called a derived class.
- Inheritance is a technique of code reuse.
- The capability of a class to derive properties and characteristics from another class is called Inheritance.



- Sub Class: The class that inherits properties from another class is called Sub class or Derived Class.
- Super Class: The class whose properties are inherited by sub class is called Base Class or Super class.

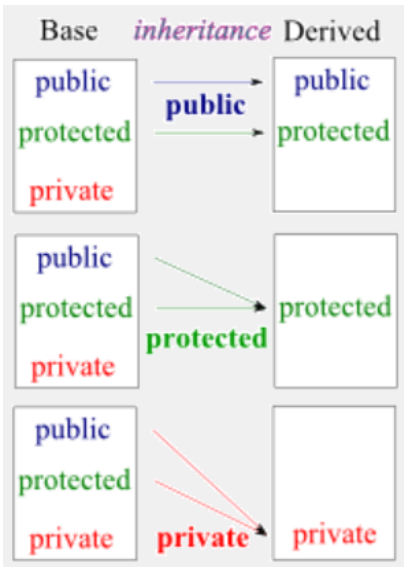
Syntax:

```
class subclass_name : access_mode base_class_name
{
  //body of subclass
};
```

- Here, subclass_name is the name of the sub class, access_mode is the mode in which you want to inherit this sub class for example: public, private etc. and base_class_name is the name of the base class from which you want to inherit the sub class.

Note: private member of the base class will never get inherited in the sub class.

Modes of Inheritance



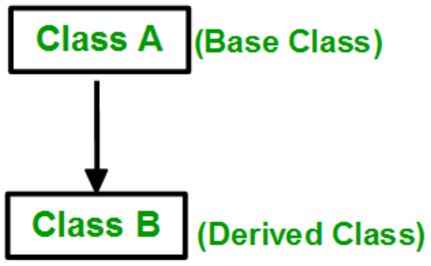
1. **Public mode**: If we derive a sub class from a public base class. Then the public member of the base class will become public in the derived class and protected members of the base class will become protected in derived class. Private members of the base class will never get inherited in sub class.
2. **Protected mode**: If we derive a sub class from a Protected base class. Then both public member and protected members of the base class will become protected in derived class. Private members of the base class will never get inherited in sub class.
3. **Private mode**: If we derive a sub class from a Private base class. Then both public member and protected members of the base class will become Private in derived class. Private members of the base class will never get inherited in sub class.

Types of Inheritance in C++

➢ Forms of Inheritance

1. Single Inheritance
2. Multiple Inheritance
3. Hierarchical Inheritance
4. Multilevel Inheritance
5. Hybrid Inheritance (also known as Virtual Inheritance)

Single Inheritance: In single inheritance, a class is allowed to inherit from only one class. i.e. one sub class is inherited by one base class only.

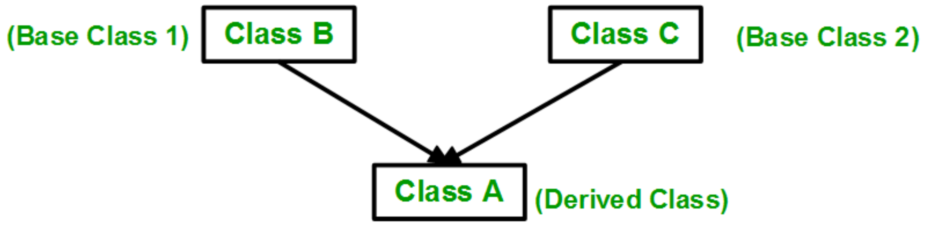


```
Syntax:
class subclass_name : access_mode base_class
{
  //body of subclass
};
```

```
// C++ program to explain  Single inheritance
#include <iostream>
using namespace std;
 // base class
class Vehicle {
  public:
    Vehicle() {
      cout << "This is a Vehicle" << endl;
    }
};

 // sub class derived from two base classes
 class Car: public Vehicle{
 };

int main()
{
    // creating object of sub class will
    // invoke the constructor of base classes
    Car obj;
    return 0;
}
```

Output:
```
This is a vehicle
```

Multiple Inheritance: Multiple Inheritance is a feature of C++ where a class can inherit from more than one classes. i.e one **sub class** is inherited from more than one **base classes**.



Syntax:

```
class subclass_name : access_mode base_class1, access_mode base_class2, ....
{
  //body of subclass
};
```

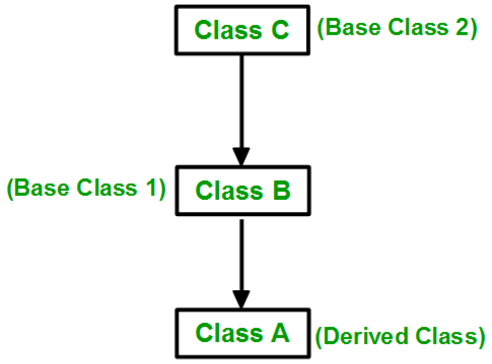
Here, the number of base classes will be separated by a comma (', ') and access mode for every base class must be specified.

```
// C++ program to explain  multiple inheritance
#include <iostream>
using namespace std;
// first base class
class Vehicle {
  public:
    Vehicle()  {
      cout << "This is a Vehicle" << endl;
    }
};

// second base class
class FourWheeler {
  public:
    FourWheeler() {
      cout << "This is a 4 wheeler Vehicle" << endl;
    }
};

// sub class derived from two base classes
class Car: public Vehicle, public FourWheeler {

};

// main function
int main(){
    // creating object of sub class will
    // invoke the constructor of base classes
    Car obj;
    return 0;
}
```

- Output:

```
This is a Vehicle
This is a 4 wheeler Vehicle
```

Multilevel Inheritance: In this type of inheritance, a derived class is created from another derived class.

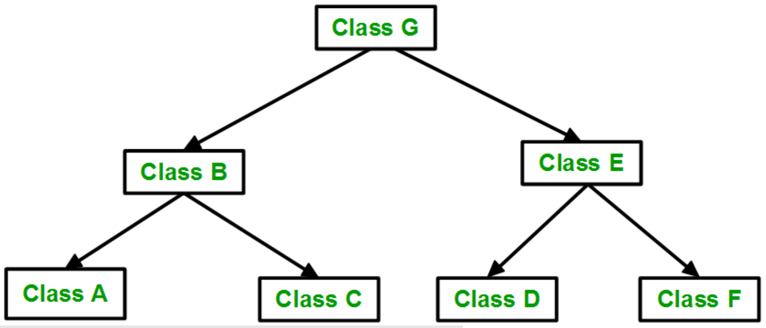


```
// C++ program to implement  Multilevel Inheritance
#include <iostream>
using namespace std;
 // base class
class Vehicle{
  public:
    Vehicle(){
          cout << "This is a Vehicle" << endl;
    }
};
class fourWheeler: public Vehicle{
     public:
          fourWheeler(){
              cout<<"Objects with 4 wheels are vehicles"<<endl;
           }
};
// sub class derived from two base classes
class Car: public fourWheeler{
    public:
         car(){
                cout<<"Car has 4 Wheels"<<endl;
          }
};

// main function
int main(){
    //creating object of sub class will
    //invoke the constructor of base classes
    Car obj;
    return 0;
}
```

- output:

```
This is a Vehicle
Objects with 4 wheels are vehicles
Car has 4 Wheels
```

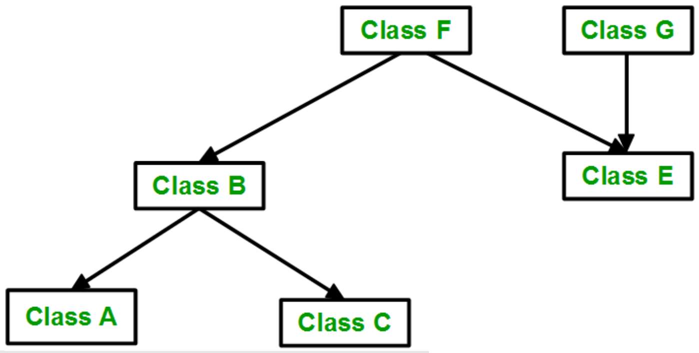
Hierarchical Inheritance: In this type of inheritance, more than one sub class is inherited from a single base class. i.e. more than one derived class is created from a single base class.



```
// C++ program to implement
// Hierarchical Inheritance
#include <iostream>
using namespace std;

// base class
class Vehicle {
      public:
            Vehicle() {
                  cout << "This is a Vehicle" << endl;
             }
};


// first sub class
class Car: public Vehicle{
 };

// second sub class
class Bus: public Vehicle{
 };

// main function
int main(){
   // creating object of sub class will
   // invoke the constructor of base class
   Car obj1;
   Bus obj2;
   return 0;
}
```

- Output: `This is a Vehicle  This is a Vehicle`

Hybrid (Virtual) Inheritance: Hybrid Inheritance is implemented by combining more than one type of inheritance. For example: Combining Hierarchical inheritance and Multiple Inheritance. Below image shows the combination of hierarchical and multiple inheritance:



```
// C++ program for Hybrid Inheritance
#include <iostream>
using namespace std;
 // base class
class Vehicle {
          public:
                  Vehicle() {
                      cout << "This is a Vehicle" << endl;
                   }
};

//base class
class Fare{
  public:
          Fare() {
               cout<<"Fare of Vehicle\n";
          }
};

// first sub class
class Car: public Vehicle{
 };

// second sub class
class Bus: public Vehicle, public Fare{
  };

// main function
int main(){
  // creating object of sub class will
  // invoke the constructor of base class
  Bus obj2;
  return 0;
}
```

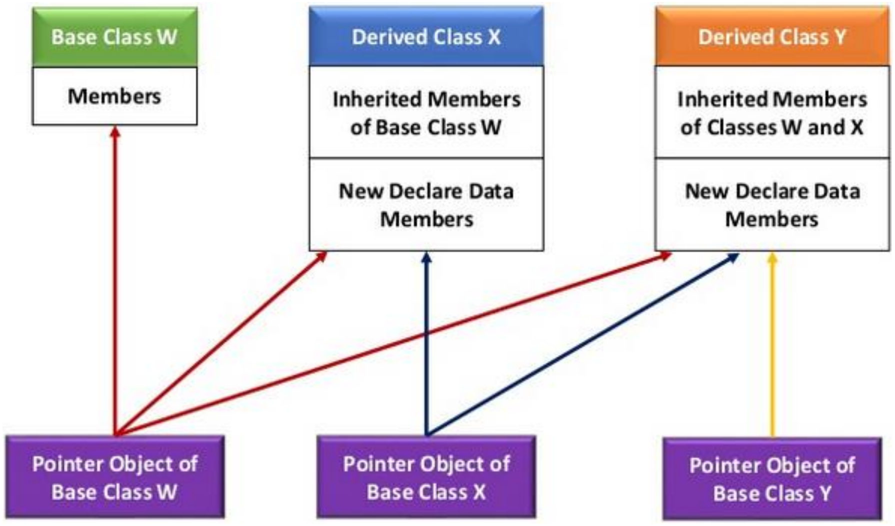
Pointers and references to the base class of derived objects

In the previous chapter, you learned all about how to use inheritance to derive new classes from existing class, now we are going to focus on one of the most important and powerful aspects of inheritance -- virtual functions.
But before we discuss what virtual functions are, let's first set the table for why we need them.
From inheritance we learned that when you create a derived class, it is composed of multiple parts: one part for each inherited class, and a part for itself.

For example, here's a simple case:

```
class Base{
protected:
   int m_value;

public:
   Base(int value): m_value(value)
   {
   }

   const char* getName() {
   return "Base";
}
   int getValue(){
 return m_value;
   }
};

class Derived: public Base
{
public:
   Derived(int value) : Base(value)
   {
   }

   const char* getName() { return "Derived"; }
   int getValueDoubled() { return m_value * 2; }
};
```

In inheritance, the properties of existing classes are extended to the new classes. The new classes that can be created from the existing baseclassare called as derivedclasses.The inheritance provides the hierarchical organization of classes. It also provides the hierarchical relationship between two objects and indicates the shared properties between them. All derived classes inherit properties from the common base class. Pointers can be declared to the point base or derived class.
Pointers to objects of the base class are type compatible with pointers to objects of the derived class. A base class pointer can point to objects of both the base and derived class. In other words, a pointer to the object of the base class can point to the object of the derived class; whereas a pointer to the object of the derived class cannot point
to the object of the base class, as shown in Figure.


Base Class W
Members
Derived Class X
Inherited Members of Base Class W
New Declare Data Members
Derived Class Y
Inherited Members of Classes W and X
New Declare Data Members
Pointer Object of Base Class W
Pointer Object of Base Class X
Pointer Object of Base Class Y

```
#include<iostream.h>
#include<constream.h>
using namespace std;
class W{
        protected:
              int w;
        public:
              W(intk){w=k;}
        Void show(){
              cout<<"\nInbaseclassW";
              cout<<"\nW="<<w;
              }
};

Class X: public W{
        protected:
              int x;
        public:
              X (int j ,int k):W(j) {x=k; }
              void show(){
                    cout<<"\n In class X";
                    cout<<"\n w="<<w;
                    cout<<"\n x="<<x;
               }
};

class Y: public X{
        public:
        int y;
};

void main(){
        W *b;
        b = new W(20); // pointer to class W
        b->show();
        delete b;
        b = new X(5,2); // pointer to class X
        b->show();
        ((X*)b)->show();
        delete b;
        X x(3,4);
        X *d=&x;
        d->show();
}
```

- Output:

```
In base class W
W=20
In base class W
W=5
In class X
w=5
x=2
In class X
w=3
x=4
```

In the above program, the class ***W*** is a base class. The **X** is derived from ***W***, and class Y is derived from class **X**. Here, the type of inheritance is multilevel inheritance. The variable ****b*** is a pointer object of the class **W**. The statement ***b=new W(20);***creates a nameless object and returns its address to the pointer ***b***. The ****/show()/**** function is invoked by the pointer object b. Here, the ****/show()/**** function of base class **W** is invoked. Using ***/delete/**//****operator, the pointer ***b*** is deleted.

The statement ***b=new X(5,2);***creates a nameless object of class ***X*** and assigns its address to the base class pointer b. Here, it should be noted that the object bis a pointer object of the base class, and it is initialized with the address of the derived class object. Again, the pointer b invokes the function ***show()***of the base class and not the function of class ***X*** (derived class.). To invoke the function of the derived class ***X***, the following statement is used:

((X*)b)->show();

In the above statement, typecasting (upcasting) is used. The upcasting forces the object of class W to behave as if it were the object of class X. This time, the function show()of class X (derived class) is invoked. The process of obtaining the address of a derived class object, and treating it as the address of a base class object is known as upcasting/./

The statement ***X x(3,4);***creates object x of class ***X***. The statement ***X*d=&x;*** declares the pointer object d of the derived class ***X*** and assigns the address of ***x*** to it. The pointer object d invokes the derived class function ***show()***.

Here are some more examples to illustrate the concept :

```
#include <iostream>
using namespace std;
class Animal{
                protected:
                         string m_name;
    // We're making this constructor protected because we don't want people creating Animal objects directly,
    // but we still want derived classes to be able to use it.
                 Animal(:string name): m_name(name){}
                public:
                          :string getName() { return m_name; }
                         const char* speak() { return "???"; }
};

class Cat: public Animal{
                public:
                         Cat(:string name): Animal(name){}
                          const char* speak() { return "Meow"; }
};

class Dog: public Animal{
                public:
                          Dog(string name): Animal(name){}
                          const char* speak() { return "Woof"; }
};

int main(){
   Cat cat("Fred");
   cout << "cat is named " << cat.getName() << ", and it says " <<
   cat.speak() << '\n';

   Dog dog("Garbo");
   cout << "dog is named " << dog.getName() << ", and it says " <<
   dog.speak() << '\n';

   Animal *pAnimal = &cat;
   std::cout << "pAnimal is named " << pAnimal->getName() << ", and it says " << pAnimal->speak() << '\n';

   pAnimal = &dog;
   std::cout << "pAnimal is named " << pAnimal->getName() << ", and it says " << pAnimal->speak() << '\n';

   return 0;
}
```

This produces the result:

```
cat is named Fred, and it says Meow
dog is named Garbo, and it says Woof
pAnimal is named Fred, and it says ???
pAnimal is named Garbo, and it says ???
```

We see the same issue here. Because pAnimal is an Animal pointer, it can only see the Animal portion of the class. Consequently, pAnimal->speak() calls Animal::speak() rather than the Dog::Speak() or Cat::speak() function.

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

class Weapon{
   public:
      void loadFeatures()
        { cout << "Loading weapon features.\n"; }
};

class Bomb : public Weapon{
   public:
      void loadFeatures()
        { cout << "Loading bomb features.\n"; }
};

class Gun : public Weapon{
   public:
      void loadFeatures()
        { cout << "Loading gun features.\n"; }
};

int main()
{
   Weapon *w = new Weapon;
   Bomb *b = new Bomb;
   Gun *g = new Gun;

   w->loadFeatures();
   b->loadFeatures();
   g->loadFeatures();

   return 0;
}
```

Output

```
Loading weapon features.
Loading bomb features.
Loading gun features.
```

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

class Weapon{
   public:
     void features()
       { cout << "Loading weapon features.\n"; }
};

class Bomb : public Weapon{
   public:
      void features()
        { cout << "Loading bomb features.\n"; }
};

class Gun : public Weapon{
   public:
      void features()
        { cout << "Loading gun features.\n"; }
};

class Loader{
  public:
    void loadFeatures(Weapon *weapon)
    {
       weapon->features();
    }
};

int main(){
   Loader *l = new Loader;
   Weapon *w;
   Bomb b;
   Gun g;
   w = &b;
   l->loadFeatures(w);
   w = &g;
   l->loadFeatures(w);
   return 0;
}
```

Output

```
Loading weapon features.
Loading weapon features.
Loading weapon features.
```

Virtual Functions

A virtual function a member function which is declared within base class and is re-defined (Overriden) by derived class.When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class’s version of the function.

- Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call.
- They are mainly used to achieve Runtime polymorphism
- Functions are declared with a **virtual** keyword in base class.
- The resolving of function call is done at Run-time.

Rules for Virtual Functions:

1. They Must be declared in public section of class.
2. Virtual functions cannot be static and also cannot be a friend function of another class.
3. Virtual functions should be accessed using pointer or reference of base class type to achieve run time polymorphism.
4. The prototype of virtual functions should be same in base as well as derived class.
5. They are always defined in base class and overridden in derived class. It is not mandatory for derived class to override (or re-define the virtual function), in that case base class version of function is used.
6. A class may have virtual destructor but it cannot have a virtual constructor.

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

class Weapon{
   public:
     virtual void features() {
              cout << "Loading weapon features.\n"; }
};

class Bomb : public Weapon{
   public:
      void features() {
             cout << "Loading bomb features.\n"; }
};

class Gun : public Weapon{
   public:
      void features() {
              cout << "Loading gun features.\n"; }
};

class Loader{
  public:
    void loadFeatures(Weapon *weapon) {
      weapon->features();
   }
};

int main(){
   Loader *l = new Loader;
   Weapon *w;
   Bomb b;
   Gun g;

   w = &b;
   l->loadFeatures(w);

   w = &g;
   l->loadFeatures(w);

   return 0;
}
```

Pure Virtual Functions and Abstract Classes in C++:

Sometimes implementation of all function cannot be provided in a base class because we don’t know the implementation. Such a class is called abstract class. For example, let Shape be a base class. We cannot provide implementation of function draw() in Shape, but we know every derived class must have implementation of draw(). Similarly an Animal class doesn’t have implementation of move() (assuming that all animals move), but all animals must know how to move. We cannot create objects of abstract classes.

```
// pure virtual functions make a class abstract
#include<iostream>
using namespace std;

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

int main(void)
{
   Test t;
   return 0;
}
```

Output:

```
Compiler Error: cannot declare variable 't' to be of abstract
 type 'Test' because the following virtual functions are pure
within 'Test': note:   virtual void Test::show()
```

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

// An abstract class with constructor
class Base{
            protected:
                   int x;
            public:
                  virtual void fun() = 0;
                  Base(int i) { x = i; }
};

class Derived: public Base{
                   int y;
            public:
                   Derived(int i, int j):Base(i) { y = j; }
                   void fun() { cout << "x = " << x << ", y = " << y; }
};

int main(void){
  Derived d(4, 5);
  d.fun();
  return 0;
}}
```

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

class Box {
  public:
            // Constructor definition
            Box(double l = 2.0, double b = 2.0, double h = 2.0) {
                  cout <<"Constructor called." << endl;
                  length = l;
                 breadth = b;
                 height = h;
             }

            double Volume() {
                  return length * breadth * height;
             }

             int compare(Box box) {
                  return this->Volume() > box.Volume();
             }
     private:
             double length;     // Length of a box
             double breadth;    // Breadth of a box
            double height;     // Height of a box
};
```

```
int main(void) {
                Box Box1(3.3, 1.2, 1.5);    // Declare box1
                Box Box2(8.5, 6.0, 2.0);    // Declare box2

               if (Box1.compare(Box2)) {
                         cout << "Box2 is smaller than Box1" <<endl;
                                                                    }
                else {
                        cout << "Box2 is equal to or larger than Box1" <<endl;
                         }

                return 0;
}
```

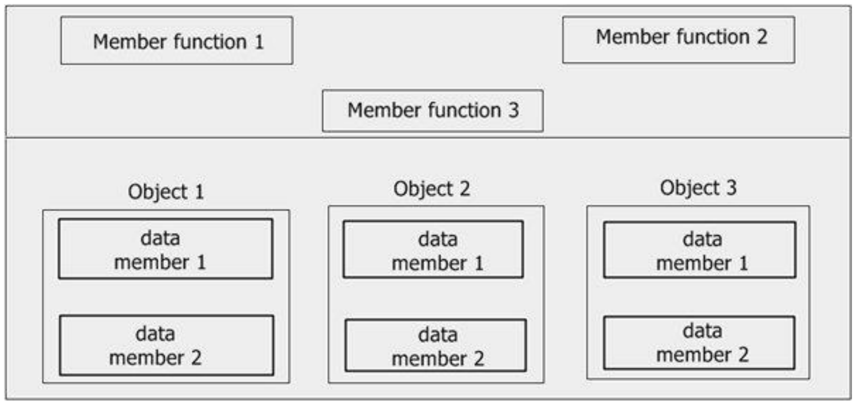
Output:

```
x = 4, y = 5
```

The This Pointer

- Every object has a special pointer "this" which points to the object itself.
- This pointer is accessible to all members of the class but not to any static members of the class.
- Can be used to find the address of the object in which the function is a member.
- Presence of this pointer is not included in the size of calculations.

We know that while defining a class the space is allocated for member functions only once and separate memory space is allocated for each object, as shown in figure



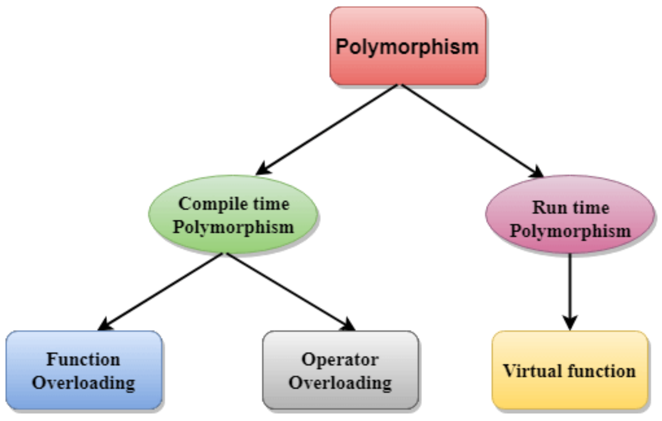
With the above shown allocation there exists a serious problem that is which object's data member is to be manipulated by any member function. For example, if memberfunc2 is responsible for modifying the value of data member1 and we are interested in modifying the value of datamember1 of object3. In the situation like it, how to decide the manipulation of which object's datamember1?, The this pointer is an answer to this problem. The this is a pointer that points to that object using which the function is called.

The This pointer is automatically passed to a member function when it is called. The following program illustrates the above mentioned concept :

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

class MyClass {
      int data;
   public:
      void SetData(int data);
      int GetData() { return data; };
};

// Same name for function argument and class member
// this pointer is used to resolve ambiguity
void MyClass::SetData(int data) {
   this->data = data;
}

int main()
{
   MyClass a;
   a.SetData(100);
   cout << a.GetData() << endl;
}
```

Polymorphim :

Polymorphism means 'one name multiple forms'. Runtime polymorphism can be achieved by using virtual functions. The polymorphism implementation in C++ can be shown as in figure.



Static Polymorphism or Compile Time Polymorphism:

It means existence of an entity in various physical forms simultaneously. Static polymorphism refers to the binding of functions on the basis of their signature (number, type and sequence of parameters). It is also called early binding because the calls are type and sequence of parameters). It is also called early binding because the calls are already bound to the proper type of functions during the compilation of the program. For example,

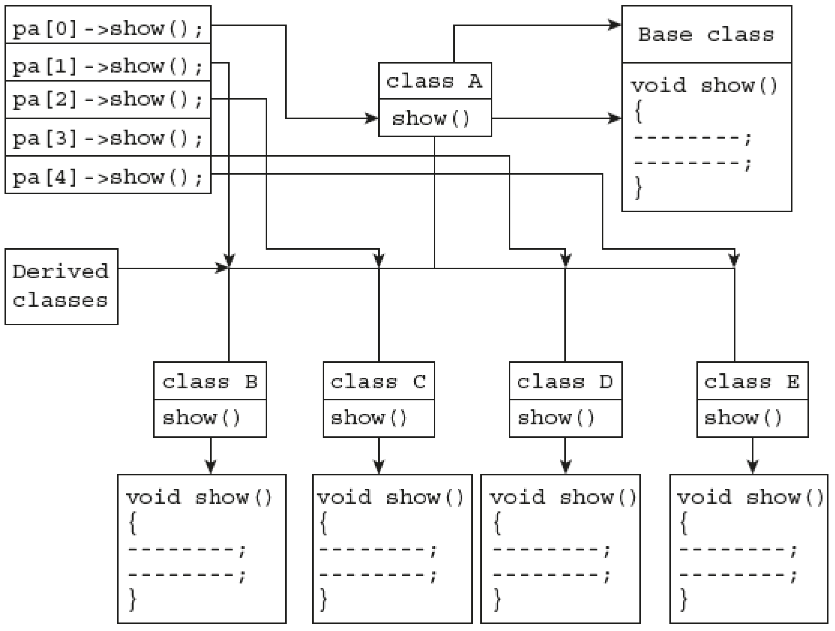
```
Void volume (int); //prototype
Void volume (int,int,int); //prototype
```

When the function volume ( ) is invoked, the passed parameters determine which one to be executed. This resolution takes place at compile time.

Dynamic Polymorphism: It means change of form by entity depending on the situation. A function is said to exhibit dynamic polymorphism if it exists in various forms, and the resolution to different function calls are made dyanamically during execution time. This feature makes the program more flexible as a function can be called, depending on the context.

Static And Dynamic Binding: As stated earlier the dynamic binding is more flexible, and the static binding is more efficient in certain cases. Statically bound functions do not require run-time search, while the dynamic function calls need it. But in case of dynamic binding, the function calls are resolved at execution time and the user has the flexibility to alter the call without modifying the source code. For a programmer, efficiency and performance are more important, but to the user, flexibility and maintainability are of primary concern. So a trade-off between the efficiency and flexibility can be made.

Array of Pointers: With reference to late or dynamic binding; that is, the selection of an entity is decided at run time. In class hierarchy, methods with similar names can be defined, which perform different tasks, and then, the selection of the appropriate method is done using dynamic binding. Dynamic binding is associated with object pointers. Thus, addresses of different objects can be stored in an array to invoke functions dynamically. The following program explains this concept:



A program to create array of pointers. Invoke functions

```
using array objects.*
#include<iostream.h>
#include<constream.h>
classA{
		public:
			virtual void show(){ cout <<"A\n";}};
classB:publicA{
		public:
			voidshow(){cout<<"B\n";}};
classC:publicA{
		public:
			voidshow(){cout<<"C\n";}};
classD:publicA{
		public:
			voidshow(){cout<<"D\n";}};
classE:publicA{
		public:
			voidshow(){cout<<"E";}
};
voidmain(){
		clrscr();
		Aa;
		B b;
		C c;
		D d;
		E e;
		A *pa[]={&a,&b,&c,&d,&e};
		for ( int j=0;j<5;j++)
			pa[j]->show();
  }
```

In the above program, classAis a base class. The classes B, C, D,and E are classes derived from class A. All these classes have a similar function show(). In function main(), a, b, c, d,and e are objects of classes A,B,C,D,and E respectively. The function show()of the base class is declared virtual. An array of pointer *pa is declared, and it is initialized with addresses of the base and derived class objects; that is, a, b, c, d and e. Using forloop and array, each object invokes function show(). The output is as shown above. If the base class function show()is non-virtual, then the very time function show() of base class is executed. Figure illustrates this concept more clearly.

VTU Question Paper Questions :

1. Discuss base class and derived class with suitable examples?
2. What is inheritance? List its advantages.
3. Explain the visibility inheritance modes. Give examples.
4. Define class inheritance. How Public, Private and Protected inheritance ate implemented. Give examples.
5. Compare multiple-inheritance with multilevel-inheritance?
6. Explain multiple inheritance with constructor and destructor operation executed with example.
7. What is abstract class? Give examples.
8. What is Hybrid Inheritance? Explain the diamond problem of inheritance in C++ with suitable examples.
9. What is an abstract class? Write the advantages with an example program.
10. Give the significance of "this" pointer with program.
11. Demonstrate the working of pointers as objects with relevant examples.
12. Differentiate virtual and pure virtual functions.
13. List the rules of virtual function in C++.