# Unit → 6

# Virtual function, Polymorphism, and miscellaneous C++ features.

Date.
Page No.

## Virtual Function:

A virtual function a member function which is declared within a base class and is re-defined (Overriden) by a derived class.

When we use the same function name in both base class and derived class the function in the base class is declared by using keyword virtual. When virtual functions are used different functions can be executed by the same function call statement.

### Rules

Virtual functions ensure that the correct function is called for an object. They are mainly used to achieve runtime polymorphism. Virtual functions cannot be static and also cannot be a friend function of another class. Virtual functions should be accessed using pointer or refrence of base class type to achieve run time polymorphism. The prototype of virtual functions should be same in base as well as derived class. They are always defined in base class and overridden in derived class. A class may have virtual destructor but it cannot have a virtual constructor.

Consider the following simple program showing run-time behaviour of virtual functions.

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

class base { public:
        virtual void print(){
cout << "Print base class" << endl; }

void show(){
        cout << "Show base class" << endl;
};
```

**Polymorphism** → Polymorphism means that same thing or name exists in many forms to perform various actions.

**Type** → 1) Compile time (or static) → achieved using overloading.
2) Runtime (or Dynamic) → achieved using overriding.

```
class derived : public base { public:
                                void print (){
cout << "Print derived class" << endl; }

                                void show (){
cout << "Show derived class" << endl; }
};

int main (){
        base *bptr;
        derived d;
        bptr = &d;

    bptr -> print(); //virtual function, binded at runtime.
bptr -> show(); //Non-virtual function, binded
                                at compile time.
}
```

**Output:**

```
Print derived class
Show base class
```

**Explanation:** Runtime polymorphism is achieved only through a pointer (or refence) of a base class type. Also a base class pointer can point to the objects of base class as well as to the objects of derived class. In above code, base class pointer 'bptr' contains the address of object 'd' of derived class.

**Note:** If we have created a virtual function in the base class and it is being overridden in the derived class then we don't need virtual keyword in the derived class, functions are automatically considered as virtual functions in the derived class.

Date.
Page No.

## ⊛ Difference between normal member function accessed with pointers and virtual member function accessed with pointer

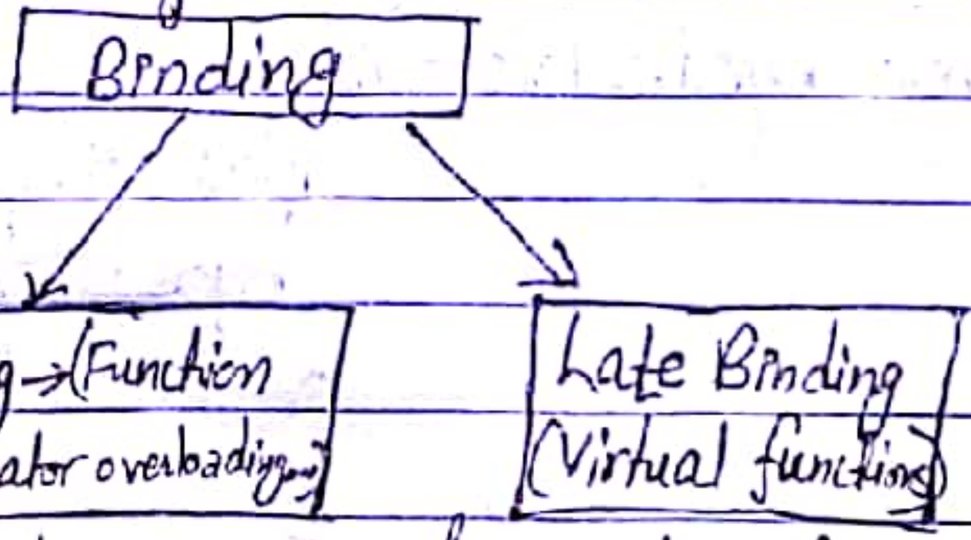
Ans: When a member function is virtual, the function called is determined by the actual most derived type of the object named by the expression left of the dot (.) or pointed to by the expression left of the arrow (→). This is called the "dynamic type".

In contrast, virtual member functions are resolved ~~stat~~ dynamically (at run-time).

When a member function is not virtual, the function ~~it~~ called is determined only by the type of the expression to the left of dot (.) or arrow (→) operator. This is called the "static type".

## ⊛ Early binding and Late binding in C++

Binding refers to the process of converting identifiers such as variable names into addresses. Binding is done done for each variable and functions.




Early Binding (compile-time polymorphism) → As the name indicates compiler (or linker) directly associate an address to the function call. It replaces the call with a machine language instruction that tells the mainframe to leap to the address of the function.

Date.
Page No.

## Late Binding: (Run time polymorphism)

In this, the compiler adds code that identifies the kind of object at runtime then matches the call with the right function definition. This can be achieved by declaring a virtual function. Below is the program to illustrate late binding.

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

class Base { public:
        virtual void show(){
cout<<"In Base \n"; }
    };

class Derived :public Base{ public:
        void show(){
    cout<<"In Derived \n"; }
    };

int main(void){
        Base *bp=new Derived;
        bp->show(); //Run-time polymorphism.
        return 0;
    }
```

Output:

```
In Derived
```

Date.
Page No.

## 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 can not provide implementation of function draw() in Shape. If we provide it then it will be meaningless while defining. Similarly an Animal class doesn't have implementation of move() (assuming that all animals move). We can not create objects of abstract class.

A pure virtual function (or abstract function) in C++ is a virtual function for which we don't have implementation, we only declare it. A pure virtual function is declared by assigning 0 in declaration. See the following example.

```
class Test   //An abstract class
{
    // Data members of class
public:
    //Pure virtual function.
    virtual void show()=0;
    // other members.
};
```

A pure virtual function is implemented by classes which are derived from a Abstract class.

Upcasting → Process of assigning address of object of derived class to base class is called pointer to base class or upcasting.

Date:
Page No.

## ⊛ Virtual destructor, virtual base class:

Deleting a derived class object using a pointer to a base class that has a non-virtual destructor results in undefined behaviour. To correct this situation, the base class should be defined with a virtual destructor. For example following program results in undefined behaviour.

```
class Base{ public:
    ~Base(){
        cout<<"Base Destructor\n";
    }
};

class Derived: public Base{ public:
    ~Derived(){
        cout<<"Derived Destructor\n";
    }
};

int main(){
    Base *b = new Derived //Upcasting
    delete b;
}
```

Output:

```
Base Destructor
```

In the above example, delete b will only call the Base class destructor, which is undesirable because, then the object of Derived class remains undestructed, because its destructor is never called, which may result in memory leak.

Now, lets see, what happens when we have virtual destructor in the base class.

Date. _____
Page No._____

```
class Base { public:
        virtual ~Base(){
        cout << "Base Destructor \n";
        }
};
class Derived : public Base { public:
        ~Derived(){
        cout << "Derived Destructor";
        }
};
int main(){
        Base *b = new Derived; // Upcasting
        delete b;
}
```

Output:
Derived Destructor
Base Destructor

(*) Static function: Just like the static data members or ~~and~~ static variables inside the class, static member function also does not depend on object of class. When a variable is declared as static the allocated space is seperate for each. There can not be multiple copies of same static variables for different objects. We are allowed to invoke a static member function using the object and the '.' operator but it is recomended to invoke the static members using the class name and the scope resolution operator. Static member functions are alloweded to access only the static data members or other static member functions, they can not access the non-static data members or member functions of class.

Date.
Page No.

```
// C++ Program to demonstrate static member function
// in a class

#include<iostream>
using namespace std;

class Check { public:
        static void printMsg() //Static m.f
        {
        cout<<" Welcome to Check";
        }
};
int main (){
        Check::printMsg(); //invoking static member
}
```

(Keyword)

## ⑧ Concrete class vs. Abstract class

There are two main types of classes; Abstract class and Concrete class. The main difference between the two ~~axies~~ arises from the level of implementation of their method functionalities. Concrete classes are regular classes, where all methods are completely implemented. An abstract class is exactly what its name suggests. It is where the functions are not defined. i.e they are abstract. A concrete class is derived from base class.

An abstract class can never be final, as it has no defined functions. Hence each program must have a concrete class, in order to tell it which functions to implement and how.

Date.
Page No.

## ⊛ Pointer to base class:

The object is of class type and can be used as its own type. The process of taking the address of an object and treating it as the address of the base type is called upcasting. The pointer to the desired class but the pointer to the base class is not type compatible with a pointer to its derived class.

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

class shape { protected:
                float l,b,p;
              public:
                void setdata (int x, int y) {
                    l=x; b=y; }
};

class square: public shape { public:
                void peri(){
                    p=4*l;
                    cout<<"The perimeter is"<<p<<endl;
                }
};

int main (){ shape *bp;
             square s;
             bp=&s;
             s.peri();
             bp->setdata (2,2);
           }
```

If bp→peri() is used it can't be accessed because peri is not the member of shape or peri is not inherited in shape.

Date.
Page No.

## ⊛ 'This' pointer:

'This' is a keyword by C++ to represent a object that ~~inov~~ invokes member function. It is a pointer that points to the object for which this function was called.

To understand 'this' pointer, it is important to know how objects look at functions and data members of class.

i) Each object gets its own copy of the data member.
ii) All objects share single copy of member functions.

The 'this' pointer is passed as a hidden argument to all nonstatic member function calls and is available as a local variable within the body of all nonstatic functions. 'this' pointer is a constant pointer that holds the memory address of the current object. 'this' pointer is not available in static member functions as static members functions can be called without any object (with class name).

E.g. For a class X, the type of this pointer is 'X *const'.

Following are the situations where 'this' pointer is used:

i) When local variable's name is same as member's name:

```
#include<iostream>
using namespace std;
/* local variable name is same as members name */
class Test {
    private:
        int x;
    public
        void setX (int x) {
```

Date.
Page No.

// The 'this' pointer is used to retrive the object's x.
// hidden by the local variable 'x':

```
        this->x = x;
    }
    void print(){ cout<<"x="<<x<<endl; }
};
int main (){
    Test obj;
    int x=20;
    obj.setX(x);
    obj.print();
    return 0;
}
```

Output

x=20

* To return refrence to the calling object:

When a refrence to a local object is returned, the returned refrence can be used to chain function calls on a single object.

```
#include<iostream>
using namespace std;
class Test { private:
    int x, y;
public:
    Test (int x=0, int y=0) {
    this->x=x; this->y=y; }
```

Date.
Page No.

```
Test &setX(int a) { x=a; return *this; }
Test &setY(int b) { y=b; return *this; }

void print() { cout << "x=" << x << "y=" << y << endl; }

int main() {
        Test obj1(5,5);
// Chained function calls. All calls modify the same
// object as the same object is returned by refrence.
    obj.setX(10).setY(20);
        obj.print();
return 0;
    }
```

Output:

x=10 y=20