Marwadi education foundation

Prof. Meghnesh Jayswal

UNIT 6
Virtual Functions and Polymorphism

KNOWLEDGE IS THE CURRENCY
FOR THE 21st CENTURY

Objective of this Unit

- Concept of objects and pointers,
- Virtual Functions and Pure Virtual Functions ,
- this pointer,
- Virtual Constructor and Destructor

Pointer to Object

- Similar to variable, objects also have an address. A pointer can point a specified object.

```
#include<iostream.h>
#include<conio.h>
class Bill
{
    int qty;
    float price;
    float amount;
    public :
    void getdata (int a, float b, float c)
    {
        qty=a;
        price=b;
        amount=c;
    }
    void show()
    {
    cout<<“Quantity : ” <<qty <<“\n”;
    cout<<“Price : ” <<price <<“\n”;
    cout<<“Amount : ” <<amount <<“\n”;
    }
};
```

```
int main()
{
    clrscr();
    Bill s;
    Bill *ptr =&s;
    ptr->getdata(45,10.25,45*10.25);
    (*ptr).show();
    return 0;
}
```

OUTPUT

```
Quantity : 45
Price : 10.25
Amount : 461.25
```

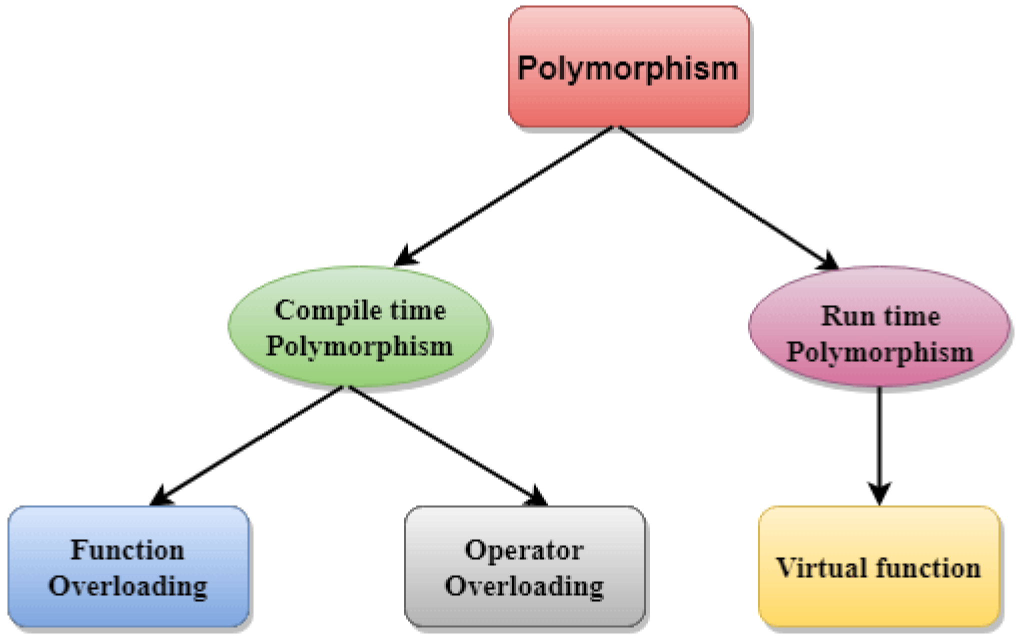
Explanation: In the above program, the class Bill contains two float and one int members. The class Bill also contains the member function getdata() and show() to read and display the data. In function main(), s is an object of class Bill, and ptr is a pointer of the same class. The address of object s is assigned to pointer ptr. Using pointer ptr with arrow operator (->) and dot operator (.), members and functions are invoked. The statements used for invoking functions are as given below.

```
ptr->getdata (45,10.25,45*10.25);
(*ptr).show();
```

Here, both the pointer declarations are valid. In the second statement, ptr is enclosed in brackets, because the dot operator (.) has higher precedence as compared with the indirection operator (*). The output of the program is as shown above.

Polymorphism

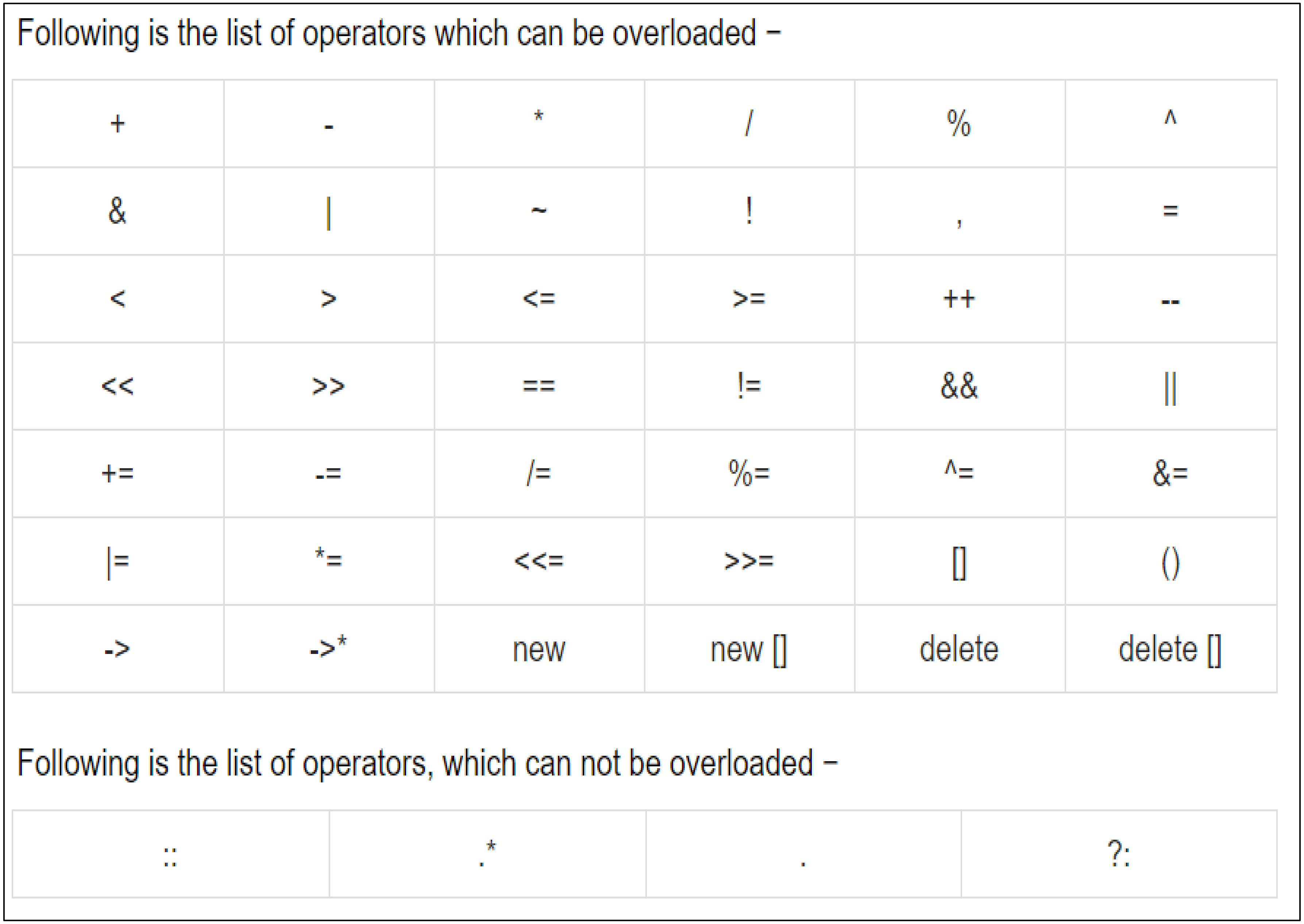
- Polymorphism means the ability to take more than one form.
- It allows a single name to be used for more than one related purpose.
- It means ability of operators and functions to act differently in different situations.
- Different types of polymorphism are



Polymorphism: Compile time

- Compile time polymorphism is function and operator overloading.

- **Function Overloading:**
  - Function overloading is the practice of declaring the same function with different signatures.
  - The same function name will be used with different number of parameters and parameters of different type.
- Overloading of functions with **different return type is not allowed.**
- **Operator Overloading**:
  - Operator overloading is the ability to tell the compiler how to perform a certain operation based on its corresponding operator's data type.
  - Like **+** performs addition of two integer numbers, concatenation of two string variables and works totally different when used with objects of time class.



Following is the list of operators which can be overloaded –

+	-	*	/	%	^
&	|	~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=
|=	*=	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []

Following is the list of operators, which can not be overloaded –

::	.*	.	?:

List of operators that cannot be overloaded

- Scope Resolution Operator (::)
- Ternary or Conditional Operator (?:)
- Member Access or Dot operator (.)
- Pointer-to-member Operator (.*)
- Object size Operator (sizeof)
- Object type Operator(typeid)
- static_cast (casting operator)
- const_cast (casting operator)
- reinterpret_cast (casting operator)
- dynamic_cast (casting operator)

Polymorphism: Run time(Dynamic Binding ,Late Binding)

- Dynamic binding is the linking of a routine or object at runtime based on the conditions at that moment.
- It means that the code associated with a given procedure call is not known until the time of the call.
- At run-time, the code matching the object under current reference will be called.

- **Virtual Function:**
  - Virtual function is a member function of a class, whose functionality can be over-ridden in its derived classes.
  - The whole function body can be replaced with a new set of implementation in the derived class.
  - It is declared as virtual in the base class using the virtual keyword.

Virtual Function

- It is a run time polymorphism.
- Virtual means existing in effect but no in reality, when virtual function are used in a program, that appears to be calling a function of one class may in reality be calling a function in different class.
- A function is said to be virtual when the same function name is used in both base and derived classes, and the keyword virtual is written before the return type of the function in the base class definition.
- **By using virtual function, we can call derived class member function through base class pointer.**
- To declare virtual function just uses keyword virtual preceding its normal function declaration in base class.
- The keyword virtual tells the compiler that it should not perform early binding(static).Instead it should be automatically install all the mechanism necessary to perform late binding.

Virtual Function

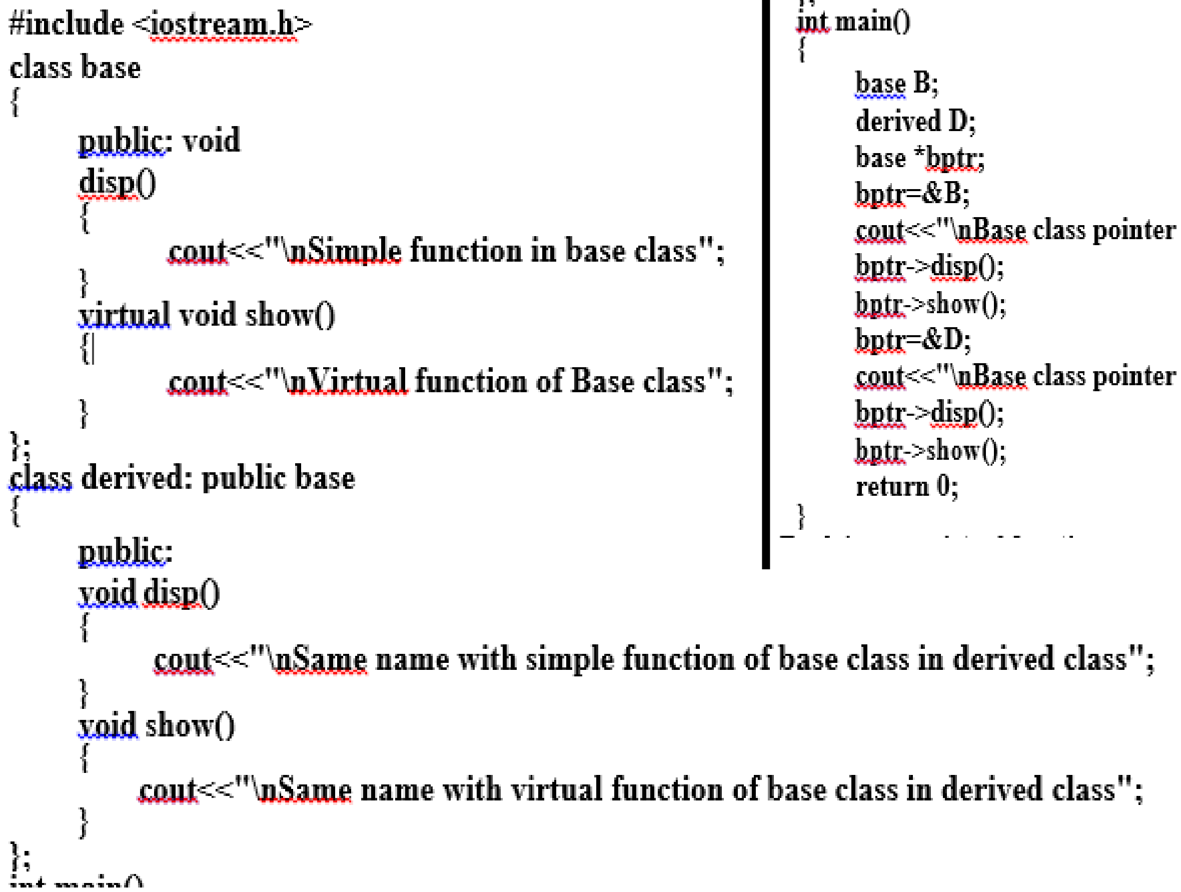
Rules for virtual function

1. The virtual functions must be member of any class.
2. They cannot be static members.
3. They are accessed by using object pointers.
4. A virtual function can be a friend of another class.
5. A virtual function in a **base class must be defined**, even though it may not be used.
6. If two functions with the same name have different prototypes, C++ considers them as overloaded functions, and the virtual function mechanism is ignored.
7. We cannot have virtual constructors, but we can have virtual destructors.

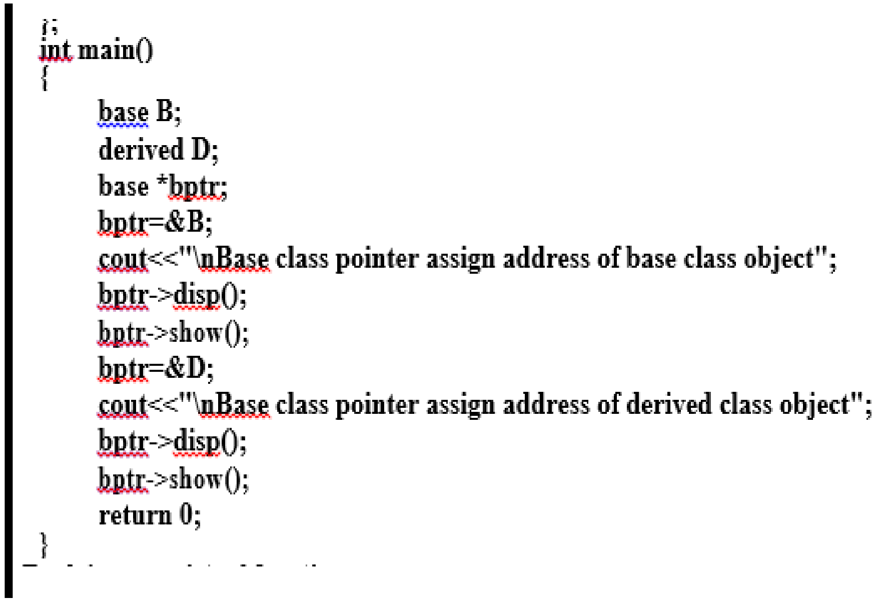
Virtual Function

8. The derived class pointer cannot point to the object of base class.
9. If a virtual function is defined in base class, it need not be necessarily redefined in the derived class. In such cases, call will invoke the base class.

Virtual Function



```
#include <iostream.h>
class base
{
        public: void
        disp()
        {
                cout<<"\nSimple function in base class";
        }
        virtual void show()
        {|
                cout<<"\nVirtual function of Base class";
        }
};
class derived: public base
{
        public:
        void disp()
        {
              cout<<"\nSame name with simple function of base class in derived class";
        }
        void show()
        {
            cout<<"\nSame name with virtual function of base class in derived class";
        }
};
int main()
```



```
};
int main()
{
        base B;
        derived D;
        base *bptr;
        bptr=&B;
        cout<<"\nBase class pointer assign address of base class object";
        bptr->disp();
        bptr->show();
        bptr=&D;
        cout<<"\nBase class pointer assign address of derived class object";
        bptr->disp();
        bptr->show();
        return 0;
}
```

Pure Virtual function

- In example, the virtual function show() in the base class never gets executes, so we can easily do away with the body of this virtual function and add a notation = 0 in the function declaration.
- Class base
- {

```
public:
        virtual void show()=0;
```

- };
- The show() function is known as pure **virtual function.** Thus a **pure virtual function is a virtual function with no body and = 0 in its declaration.**
- The = sign here has got nothing to do with assignment, the value 0 is not assigned to anything. It is used to simply tells the compiler that a function will be pure. i.e. it will not have any body.

this pointer

- A ‘**this**’ pointer is automatically passed to a function when it is called.
- It is also called as implicit(indirect) argument to all member function.

Application of ‘this’ pointer:

1. Using ‘this’ pointer any member function can find out the address of the object of which it is member.
2. Accessing data member with ‘**this’** pointer.
3. **‘this’** pointer is to return the object it points to.

- For example: S.getdata();
- Here **S** is an object and **getdata()** is a member function. So, **‘this’** pointer will point or set to the address of object **S.**
- Suppose **‘a’** is private data member, we can access it only in public member function like as follow a=50;
- We access it in public member function by using **‘this’** pointer like as follow

- this->a=50;
  - Both will work same.
- The most important **advantage** of **‘this’** pointer is, **If there is same name of argument and data member than you can differentiate it**. **By using ‘this’ pointer we can access the data member and without ‘this’ we can access the argument in same function.**

```
#include <iostream.h>

class sample

{

int a;

public:

sample()

{   a=10;   }

void disp(int a)

{

cout<<"The value of argument a="<<a;

cout<<"\nThe value of data member a="<<this->a;

}

};
```

```
int main()

{

sample S; S.disp(20); return 0;

}

/* OUTPUT

The value of argument a=20

The value of data member a=10 */
```

virtual constructors and destructors

Virtual Constructor :

- Virtual constructor is not possible in C++.
- A constructor of a class can not be virtual and if causes a syntax error.

Virtual Destructor :

- The explicit(Directly) destroying of object with the use of **delete operator** to a base class pointer to the object is performed by the destructor of the base-class is invoked on that object.
- The above process can be simplified by declaring a virtual base class destructor.
- All the derived class destructors are made virtual in spite of having the same name as the base class destructor. In case the object in the hierarchy is destroyed explicitly by using delete operator to the base class pointer to a derived object, the appropriate destructor will be invoked. Where as destructor can be virtual.

```
Base *b=new Derived();
Delete b;
```

CPP program without virtual destructor

```
#include<iostream>
using namespace std;
class base {
  public:
    base()
    { cout<<"Constructing base \n"; }
    ~base()
    { cout<<"Destructing base \n"; }
};
class derived: public base {
  public:
    derived()
    { cout<<"Constructing derived \n"; }
    ~derived()
    { cout<<"Destructing derived \n"; }
};

int main(void)
{
  derived *d = new derived();
  base *b = d;
  delete b;
  getchar();
  return 0;       }
```

Output

```
 Constructing base
Constructing derived
Destructing base
```

In this program we call only base class constructor, destructor and derived constructor. But, we cant call derived class destructor

NOTE:OUTPUT FOLLOW:
Base class constructor
Derived class constructor
Derived class dedstructor
Base class destructor

A program with virtual destructor

```
#include<iostream>
using namespace std;
class base {
  public:
    base()
    { cout<<"Constructing base \n"; }
    virtual ~base()
    { cout<<"Destructing base \n"; }
};
class derived: public base {
  public:
    derived()
    { cout<<"Constructing derived \n"; }
    ~derived()
    { cout<<"Destructing derived \n"; }
};
int main(void)
{     derived *d = new derived();
  base *b = d;
  delete b;
  getchar();
  return 0;       }
```

OUTPUT

```
Constructing base
Constructing derived
Destructing derived
Destructing base
```

Thank You