Unit

9

BINDING, POLYMORPHISM & VIRTUAL FUNCTIONS

Lesson Structure

9.0 Objective

After going through this unit you will understand:

- How binding works in C++
- Virtual Functions and Pure Virtual Functions
- Abstract class
- Pointers to Derived Class
- Object Slicing
- Working of constructor or destructor and virtual function together.

9.1 Introduction

The word polymorphism is broken down into poly which means many, and morphism means several forms. Polymorphism supports overloading concept. For early binding at compilation time, the information belonging to various overloaded member functions is to be given to the compiler. And for run time binding the deciding function is called at run time. The decision of choosing a suitable member function remains suspended in dynamic binding till run time. Pointers to the base class object are type compatible with pointers to the derived class object. The definition of base class virtual functions is to be redefined in the derived classes. Same virtual function name can be used in base class and derived class. Abstract classes are same as a frame on which new classes are designed to create a class hierarchy. Virtual functions can be invoked using a pointer or a reference. Correlation between constructor or destructor and virtual function is described in this unit.

The rest of the chapter is organized as follows. Section 9.2 describes the binding concept of C++. Section 9.3 explains pointer to derived classes objects. Section 9.4 and 9.5 talks about virtual function and pure virtual function. Section 9.6 describes the abstract class. Sections 9.7 explain virtual functions in derived class. Section 9.8 discusses about object slicing. Section 9.9 explains constructor and virtual function together. Section 9.10 is about virtual destructor. Section 9.11 deals with destructor and virtual function. And finally section 9.12 concludes the chapter with its summary.

9.2 Binding in C++

Though C++ is an object-oriented programming language, it is very much inspired by procedural language. A program in C++ is executed sequentially line by line. Each line of the source program after translation is in the form of machine language. Each line in the machine language is assigned a unique address. Similarly, when a function call is encountered by the complier during execution, the function call is translated into machine language, and a sequential address is provided. Thus, binding refers to the process that is to be used for converting functions and variables into machine language addresses. The C++ supports two types of binding: static or early binding and dynamic or late binding.

Static Binding and Dynamic Binding

Binding is defined as linking of a ‘definition of a function’ to a ‘function call’ or a linking of a ‘value’ to a ‘variable’. In the course of compilation, each ‘definition of function’ is assigned a memory address. As soon as a function call is made, control of execution of program switches to that memory address and

acquire the code of function stored at that position, this is binding of 'function calling' to 'definition of function'. Binding can be categorized as 'early (static) binding' and 'late (dynamic) binding'. If it's known prior of runtime, which function will be called or what value is assigned to a variable, then it is a 'early binding', and if it arises at the run-time then it is termed as 'dynamic binding'.

a) Static (Early) Binding

A compiler mostly encounters direct function calls. A statement that directly calls a function is known as direct function calls.

Let us look at **program below that is a simple example of a direct function call**.

```
#include <iostream>
        void show(int val)
        {
                std::cout << val;
        }
        int main()
        {
                show(8); //  direct function call
                return 0;
        }
```

The above program when compiled and executed gives the following result:

```
        8
```

We can resolve direct function calls through **Early Binding**, which is also termed as **Static Binding**. Basically, this means that the compiler directly associates the name of identifier (function name or variable name) with a machine address. We should recall that all functions have addresses that is unique to them. When the compiler come across a function calling statement, it substitutes the function call with a machine linguistic instruction that communicates to CPU a movement to the function address.

Try to understand **program which uses early binding to create a simple calculator program:**

```
#include <iostream>
        int addition(int a, int b)
        {
                return a + b;
        }
        int subtraction(int a, int b)
        {
                return a - b;
        }
        int multiplication(int a, int b)
        {
                return a * b;
        }
```

```
int main()
{
	int a;
	std::cout << "Enter a number: ";
	std::cin >> a;
	int b;
	std::cout << "Enter another number: ";
	std::cin >> b;
	int opt;
	do
	{
		std::cout << "Enter an operation (0=addition, 1=subtraction, 2=multiplication): ";
		std::cin >> opt;
	}
	while (opt < 0 || opt > 2);
	int output = 0;
	switch (opt)
	{
		// call the target function directly using early binding
		case 0: result = addition(a, b); break;
		case 1: result = subtraction(a, b); break;
		case 2: result = multiplication(a, b); break;
	}
	std::cout << "The answer is: " << output << std::endl;
return 0;
}
```

On compilation and execution, the above program gives the following result:

```
Enter a number: 6
Enter another number: 7
Enter an operation (0=addition, 1=subtraction, 2=multiplication): 2
The answer is: 42
```

Since subtraction(), multiplication() and addition() all are direct function calls. The compiler will use static binding to solve them. Compiler substitutes the function call addition() with an instruction that makes the CPU move to the function address. The same holds true for multiplication() and subtraction().

b) Dynamic (Late) Binding

In few programs, it may not be available that which function will be called till the execution (when the program is executed). This is termed as **dynamic binding** or **late binding**. In C++, one means to accomplish dynamic binding is to use function pointers. A function pointer is a type of pointer that points to a function instead of a variable. A function pointer points to a function, can be called by means of the function call operator (()) on the pointer.

First let us see an example of a direct function call. **Program below is a simple program that is an example of a direct function call.**

```
#include <iostream>
        int sum(int a, int b)
        {
                return a + b;
        }
        int main()
        {
                // create a function pointer and use it to point the sum() function
                int (*p2fn)(int, int) = sum;
                 std::cout << p2fn(6, 6) << std::endl; // sum 6 + 3
                return 0;
        }
```

The program when compiled and executed gives the following output:

```
        12
```

When we use a function pointer to call a function it is called as an indirect function call. The subsequent calculator program functions just like the calculator program seen above, but instead of a direct function call, it uses a function pointer.

Try to understand **program below which uses a function pointer to invoke a function.**

```
#include <iostream>
int addition(int a, int b)
 {
        return a + b;
 }
int subtraction(int a, int b)
{
        return a - b;
}
int multiplication(int a, int b)
{
        return a * b;
}
int main()
{
        int a;
        std::cout << "Enter a number: ";
        std::cin >> a;
        int b;
        std::cout << "Enter another number: ";
```

```
        std::cin >> b;
        int opt;
        do
         {
                std::cout<<"Enter an operation (0=addition, 1=subtraction, 2=multiplication): ";
                  std::cin >> opt;
         }
        while (op < 0 || op > 2);
        // Create a function pointer named pFcn
        int (*pFcn)(int, int);
        // Set pFcn to point to the function the user chose
        switch (opt)
         {
                  case 0: pFcn = addition; break;
                  case 1: pFcn = subtraction; break;
                  case 2: pFcn = multiplication; break;
        }
        // Call the function that pFcn is pointing to with a and b as parameters
        // This uses dynamic binding
        std::cout << "The answer is: " << pFcn(a, b) << std::endl;
        return 0;
}
```

The above program when executed will give the following output:
Enter a number: 8
Enter another number: 9
Enter an operation (0=addition, 1=subtraction, 2=multiplication): 0
The answer is: 17

Here, instead of calling the subtraction(), addition(), or multiplication() function directly, we have used pFcn (pointer to function) to point to the function we want to invoke. Next, we use this pointer to call that function. Compiler cannot use static binding to determine the function call pFcn(a, b) as it doesn't know which function pFcn will be pointing to, at the time of compilation.

Since Dynamic Binding involves an extra level of indirection, it is slightly less efficient. With Static Binding, CPU can move to the function's address directly. In case of Dynamic Binding, the program initially reads the address stored in the pointer and then moves to that address. This contains one extra step, making Late Binding slightly slower. Nevertheless, the benefit of Late Binding is that it is more flexible. This is so because decisions on which function to call do not required to be made until execution time. A comparison is shown between Early binding and Late binding in Table 9.1.

Table: 9.1 Comparison between Static Binding and Dynamic Binding

BASIS FOR COMPARISON	STATIC BINDING	DYNAMIC BINDING
Also Known as	Early Binding, compile time binding	Late Binding, run time binding

Event Occurrence	Events occur at time of compilation are "Static Binding".	Events occur at execution time are "Dynamic Binding".
Information	All information required to call a function is present at time of compilation.	All information required to call a function comes into existence at execution time.
Advantage	Efficiency	Flexibility
Time	Execution is fast	Execution is slow.
Also known as	Early Binding	Late Binding
Example	Overloaded function call, overloaded operators.	C++ Virtual functions

9.3 Pointers to Derived Class Objects

In inheritance, the properties of existing classes are extended to the new classes. The new classes that can be created from the existing base class are called as derived classes. 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 9.1.

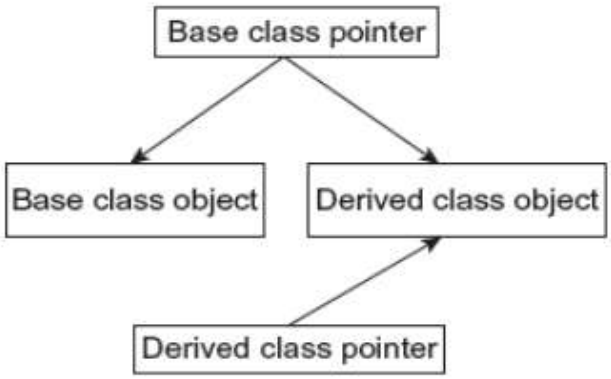

Figure 9.1: Type compatibility of base and derived class pointers

A program to access members of base and derived class using pointer objects of both classes.

```
#include<iostream.h>
#include<constream.h>
class W
{
```

```
protected:
int w;
public:
W (int k) { w=k; }
void show()
{
cout<<“\n In base class W”;
cout<<“\n W=”<<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()
{
clrscr();
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
```

Explanation: 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 b is 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(). Figure 9.2 shows a pictorial representation of the above explanation.

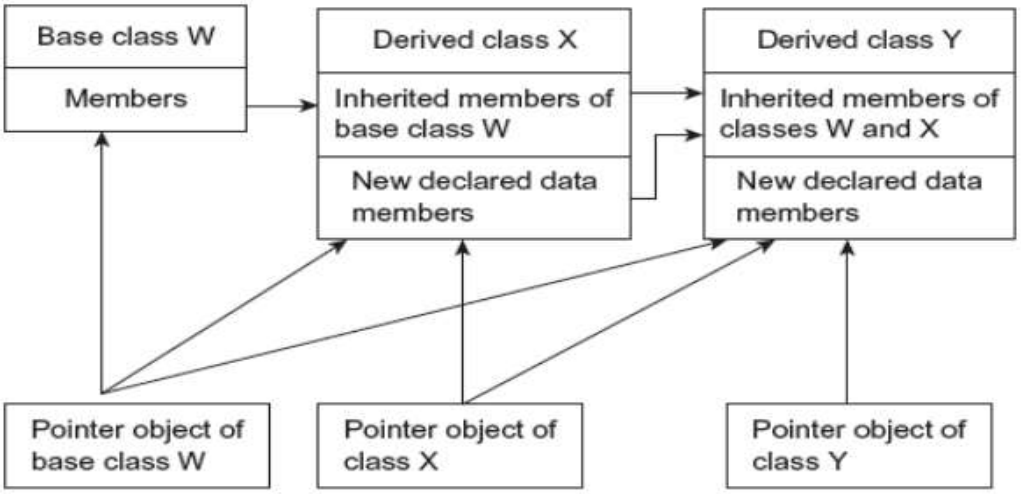

Figure 9.2: Type compatibility of base and derived class pointers

Here, the pointer of class W points to its own class as well as its derived class. The pointer of derived class X can point to its own class but cannot point to its base class.

9.4 Virtual Functions

A function of super class, which is overridden in the sub class, and that informs the compiler to execute Dynamic Binding on this function is known as Virtual Function. The keyword 'virtual' is used to make a member function of the super class as virtual.

The function call is resolved at execution time in **Late Binding**. Compiler determines the object type at execution time, and then binds the function call. Late Binding is also known as Runtime Binding or Dynamic Binding.

Let us try to understand the **problem without virtual keyword as in program below**.

```
#include<iostream>
#include<conio.h>
#include<string>
using namespace std;
class ABC
{
        public:
        void display()
         {
                cout << "This is Super class.";
         }
};
```

```
class DEF: public ABC
{
        public:
        void display()
         {
                cout << "This is Sub Class.";
         }
};

int main()
{
        ABC *a;         //Super class pointer
        DEF d;          //Sub class object
        a = &d;
        a->display();   //Early Binding Occurs
}
```

When we compile and execute the above program we get the following output:

```
This is Super class.
```

When we use Super class pointer to hold Sub class object, super class pointer or reference will always call the base version of the function. **virtual** keyword is used at time of declaration to make a super class method as virtual function. Virtual keyword will lead to Late Binding of that method.

Program below shows the usage of virtual keyword.

```
#include<iostream>
#include<conio.h>
#include<string>
using namespace std;
class ABC
 {
                public:
                virtual void output()
                 {
                        cout << "This is Base class.";
                 }
 };
class DEF: public ABC
 {
                public:
                void output()
                 {
```

```
                cout << "This is Derived Class.";
        }
};

int main()
{
        ABC *a;    // pointer to Base class
        DEF  d;   // object of Derived class
        a = &d;
        a->output();   //Late Binding Occurs
}
```

When we compile and execute the above program we get the following output:

This is Derived Class.

Late Binding takes place when **virtual** keyword is used with the Base class function. This causes the derived version of the function to be called as the base class pointer now points to object of the derived class.

We can call private function of derived class by using the keyword **virtual** with the base class pointer. Compiler checks for access specifier only at the time of compilation. It does not check at run time (when late binding occurs) whether the private or the public function is being called.

Try to understand **Program below which is another example based on the same concept**.

```
#include<iostream>
#include<string>
using namespace std;
class ABC
        {
                public:
                virtual void output()
                 {
                        cout << "This is Base class.\n";
                 }
        };

class DEF: public ABC
        {
private:
                virtual void output()
                 {
                        cout << "This is Derived class.\n";
                 }
```

```
};
int main()
{
        ABC *x;
        DEF y;
        x = &y;
        x -> output();
}
```

When we compile and execute the above program we get the following output:

```
This is Derived class.
```

a) Mechanism of Late Binding

To achieve Late Binding, compiler generates VTABLEs (virtual tables), for each class with virtual function. The virtual functions address is placed into these VTABLES. At time of an object creation of such classes, the compiler secretly attaches a pointer called virtual pointer (vptr), directing to VTABLE for that object. Hence when function call is made, compiler is capable to resolve the call by binding the appropriate function by means of the vptr.

Try to understand with the help of **program below**.

```
#include<iostream>
using namespace std;
class Base1
 {
        public:
        virtual void function_1() {cout<<"Base1 :: function_1()\n";};
        virtual void function_2() {cout<<"Base1 :: function_2()\n";};
        virtual ~Base1(){};
};
class D_1: public Base1
{
        public:
        ~D_1(){};
        virtual void function_1() { cout<<"D_1 :: function_1()\n";};
};
class D_2: public Base1
{
        public:
        ~D_2(){};
        virtual void function_2() { cout<< "D_2 :: function_2\n";};
};
int main()
{
```

```
        D_1 *d = new D_1;;
        Base1 *b = d;
        b->function_1();
        b->function_2();
        delete (b);
        return (0);
}
```

When we compile and execute the above program we get the following output:

D_1 :: function_1()

Base1 :: function_2()

To understand the program better we look at Figure 9.3 below.

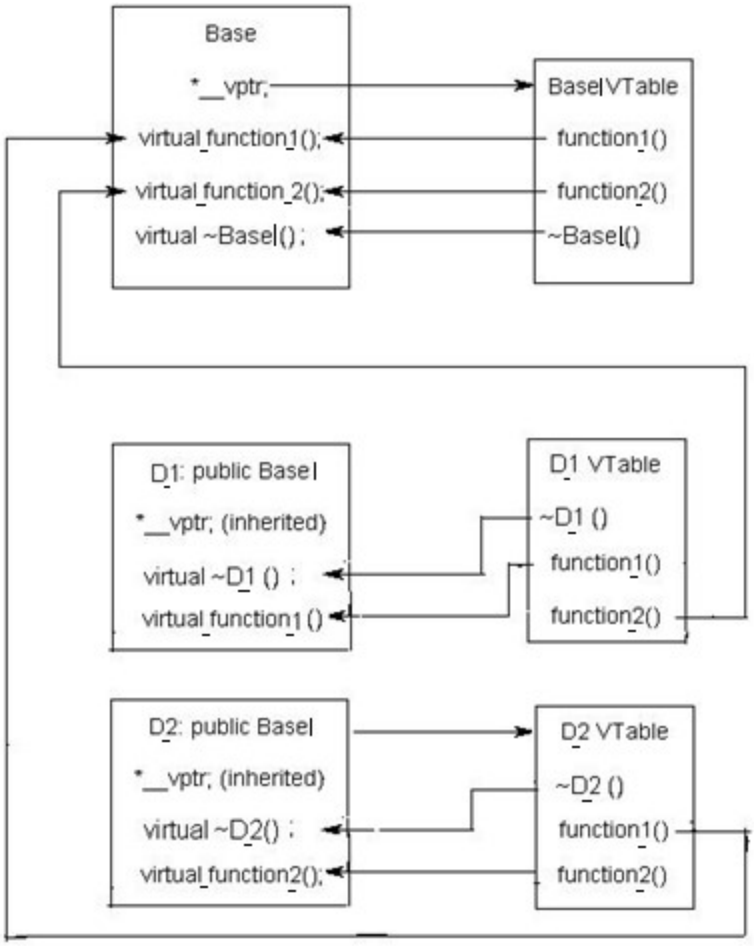

Figure 9.3: Late Binding

Here in function main b pointer gets assigned to D_1's _vptr and now starts pointing to D_1's vtable. Then calling to a function_1(), makes it's _vptr straightway calls D_1's vtable function_1() and so in turn calls D_1's method i.e. function_1() as D_1 has it's own function_1() defined it's class.

Whereas pointer b calling to a function_2(), makes it's _vptr points to D_1's vatble which in-turn pointing

to Base1 class's vtable function_2 () as shown in the diagram (as D_1 class does not have it's own definition or function_2()).

So, now calling delete on pointer b follows the _vptr - which is pointing to D_1's vtable calls it's own class's destructor i.e. D_1 class's destructor and then calls the destructor of Base1 class - this as part of when derived object gets deleted it turn deletes it's embedded base object.

That's why we must always make Base class's destructor as virtual if it has any virtual functions in it.

This is how the Dynamic binding or Run time occurs on calling virtual functions of different derived objects.

9.5 Pure Virtual Functions

Virtual functions that have no definition is known as Pure Virtual Functions. They start with a keyword **virtual** and ends with '=0'. Pure virtual function syntax is as following.

```
// An abstract class
class Test1
{
        // Data members of class
        public:
        // Pure Virtual Function
        virtual void show1() = 0;

        /* Other members */
};
```

- In the abstract class, pure virtual functions can be given a small description, which we want all the sub classes should have. However objects of an Abstract class cannot be created.
- Moreover, Pure Virtual functions should be defined external to the class definition. If we outline it inside the definition of class, complier will give a message with error. Inline pure virtual definition is illegal.

A pure virtual function is implemented by classes which are derived from an Abstract class. Take a look at **program below that shows the usage of pure virtual functions**.

```
#include<iostream>
using namespace std;
class A
{
        int m;
        public:
        virtual void learn() = 0;
        int getM()
        {
```

```
                return m;
        }
};

// This class inherits from class A and implements learn()
class B: public A
{
        int n;
        public:
        void learn()
        {
                cout << "learn() is invoked";
        }
};

int main(void)
{
        B b;
        b.learn();
        return 0;
}
```

On compilation and execution, the above program gives the following result:

```
learn() is invoked
```

9.6 Abstract Class

We call a class as ***Abstract Class*** if it contains minimum one Pure Virtual function in it. C++ allows us to use abstract classes to provide interfaces to their sub classes. A class that inherits an abstract class should define the pure virtual function(s) or else it will become an abstract class itself.

Characteristics of Abstract Class

1. Along with a pure virtual function, abstract class can also have normal functions and variables.
2. A Class will become an abstract class if it inherits an Abstract Class and doesn't define all pure virtual functions.
3. We cannot instantiate an Abstract class, but we can create pointers and references of Abstract class type.
4. Abstract classes are basically used for Upcasting, so that its interface can be used by its derived classes.

Some Interesting Facts:

1) *If a class has at least one pure virtual function, then it is abstract in nature.*

When a pure virtual function is defined in Abstract class, a slot is reserved for that function in the VTABLE (virtual table), but do not keep any address in that slot. Hence, the VTABLE information will be inadequate. As the Abstract class VTABLE is inadequate, hence the compiler will not allow the creation of object for such class. It will show an error message whenever we try to do so. The following example demonstrates this.

// a class is created as abstract by defining pure virtual functions

```
#include<iostream>
using namespace std;
 class sample
{
        int a;
public:
        virtual void display() = 0;
        int getA()
{
return a;
}
};

int main(void)
{
        sample s;
        return 0;
}
```

The above code gives the following compile time error.

Compiler Error: Variable 's' cannot declare be of abstract type 'sample' since the following virtual functions are pure inside 'sample': virtual void sample::display()

2) Abstract class type can have pointers and references.

For example, **program below works just fine.**

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

class A
{
public:
        virtual void display() = 0;
};

class B: public A
{
```

```
public:
        void display()
{
cout << "Inside Derived Class\n";
}
};
    int main(void)
{
        A *a = new B();
        a->display();
        return 0;
}
```

The above program gives the following result on execution.

    Inside Derived Class

3) The derived class becomes an abstract class, if we do not override the pure virtual function in derived class.

The subsequent example exhibits the same.

```
#include<iostream>
using namespace std;
class X
{
public:
        virtual void display() = 0;
};

class Y : public X { };

int main(void)
{
        Y y;
        return 0;
}
```

The above code gives the following compile time error.

Compiler Error: Variable 'y' cannot be declared to be of abstract type 'Y' since the following virtual functions are pure inside 'Y': virtual void X::display()

4) *An abstract class constructors can be created.*

Program below demonstrates the same.

```
#include<iostream>
```

```
using namespace std;

// An abstract class having constructor
class ABC
{
protected:
        int a;
public:
        virtual void learn() = 0;
        ABC(int b)
{
a = b;
}
};

class DEF: public ABC
{
        int c;
public:
        DEF(int b, int d):ABC(b)
{
c = d;
}
        void learn()
{
cout << "a = " << a << ", c = " << c;
}
};

int main(void)
{
        DEF d(6, 7);
        d.learn();
        return 0;
}
```

The above program when compiled and executed gives the following result:

```
a = 6, c = 7
```

9.7 Virtual Functions in Derived Class

We know that when functions are declared virtual in the super class, it is compulsory to re-define virtual functions in the sub class. The compiler creates VTABLES for the derived class and stores addresses of

functions in it. In case the virtual function is not redefined in the derived class, the VTABLE of the derived class contains the address of the base class virtual function. Thus, the VTABLE contains addresses of all functions. Thus, it is not possible for a function to exist but for its address to not be present in the VTABLE. Consider the following program, which shows the possibility of a function existing but an address not being located in the VTABLE:

A program to redefine a virtual base class function in the derived class. Also, add a new member in the derived class.

Observe the VTABLE.

```
#include<iostream.h>
#include<conio.h>
class Aa
{
public:
virtual void joy()
{
cout<<endl<<“In joy of class Aa”;
}
};
class Bb: public Aa
{
public:
void joy() {cout<<endl<<“In joy of class Bb”;}
void virtual joy2()
{
cout<<endl<<“In joy2 of class Bb”;
}
};
void main()
{
clrscr();
Aa *a1,*a2;
Aa a3;
Bb b;
a1=&a3;
a2=&b;
a1->joy();
a2->joy();
// a2->joy2(); // joy2 is not member of class Aa
}
```

OUTPUT

```
In joy of class Aa
In joy of class Bb
```

Explanation: In the above program, the base class A contains one virtual function joy(). In the derived class B, the function joy() is redefined and defines a new virtual function joy2(). Figure shows VTABLES created for the base and derived class.

a2->joy2();

The above statement will generate an error message. In the above statement, the pointer a2 is treated as only a pointer to the base class object. The function joy2() is not a member of base class A. Hence, it is not allowed to invoke the function joy2() using the pointer object a2. However, an address of the derived class is assigned to the base class pointer, and the compiler has no way to determine that we are working with a derived class object. The compiler avoids calling virtual functions present only in derived classes. In a hierarchical class organization of various levels, if it is essential to invoke a function at any level by using a base class pointer, then the function should be declared virtual in the base class. Figure 9.4 displays the VTABLE for above example.

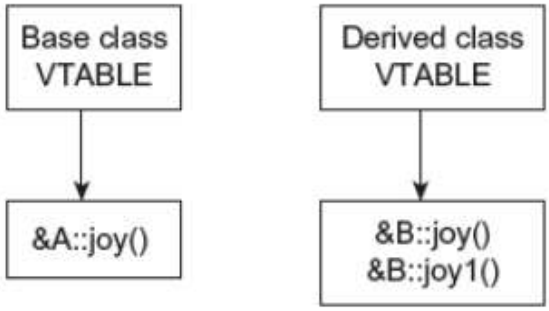

Figure 9.4: VTABLES for base and derived class

9.8 Object Slicing

Virtual functions permit us to manipulate both base and derived objects using similar member functions with no modifications. Virtual functions can be invoked using a pointer or reference. If we do so, object slicing takes place. The following program takes you to the real thing.

A program to declare reference to object and invoke functions.

```
#include<constream.h>
#include<iostream.h>
class B
{
int j;
public:
B (int jj) {j=jj;}
virtual void joy()
{
cout<<"\n In class B";
cout<<endl<<" j="<<j;
```

```
}
};
class D : public B
{
int k;
public:
D (int jj, int kk) : B (jj)
{k=kk;}
void joy()
{
B::joy();
cout<<"\n In class D";
cout<<endl<<" k="<<k;
}
};
void main()
{
clrscr();
B b(3);
D d (4,5);
B &r=d;
cout<<"\n Using Object";
d.joy();
cout<<"\n Using Reference";
r.joy();
}
```

OUTPUT

```
Using Object
In class B
j= 4
In class D
k= 5
Using Reference
In class B
j= 4
In class D
k= 5
```

Explanation: In the above program, a reference object r is created to an object d using the statement B & r = d;. The member function joy() is invoked using r and d objects. The output is similar.

A program to demonstrate object slicing.

```
#include<iostream.h>
```

```
#include<constream.h>
class A
{
public:
int a;
A() {a=10;}
};
class B: public A
{
public:
int b;
B() {a=40; b=30;}
};
void main()
{
clrscr();
A x;
B y;
cout<<" a="<<x.a <<"";
x=y;
cout<<" Now a="<<x.a;
}
```

OUTPUT

a=10 now a=40

Explanation: In the above program, class A has only one data member a and derived class B has one member b. In function main(), objects x and y of classes A and B are declared. The object x has only one member, that is, a and the object y has two members a and b. The statement x = y, that is, the derived class object, is assigned to the base class object. In such an assignment, only base, class part of the derived object is assigned to the base class object. Thus, if an object of a derived class is assigned to a base class object, the compiler allows it. However, it copies only the base class members of the object, and this process is known as object slicing.

9.9 Constructors and Virtual Function

It is possible to invoke a virtual function using a constructor. A constructor makes the virtual mechanism illegal. When a virtual function is invoked through a constructor, the base class virtual function will not be called; instead, the member function of a similar class is invoked.

A program to call virtual function through constructor.

```
#include<iostream.h>
#include<constream.h>
class B
```

```
{
int k;
public:
B (int l) {k=l;}
virtual void show() {cout<<endl<<" k="<<k;}
};
class D: public B
{
int h;
public:
D (int m, int n) : B (m)
{
h=n;
B *b;
b=this;
b->show();
}
void show()
{
cout<<endl<<" h="<<h;
}
};
void main()
{
clrscr();
B b(4);
D d(5,2);
}
```

OUTPUT

```
h=2
```

Explanation: In the above program, the base class B contains a virtual function. In the derived class D, a similar function is redefined. Both the base and derived classes contain a constructor. In the derived class constructor, the base class pointer *b is declared. We know that this pointer contains the address of the object calling the member function. Here, the this pointer holds the address of the object d. The pointer object b invokes the function show(). The derived class show() function is invoked.

Here, the object d is not fully constructed; then, how does it invoke the member function of a similar class? This is possible, because a virtual function call reaches ahead into inheritance.

9.10 Virtual destructor

We have learned how to declare virtual functions. Likewise, destructors can be declared as virtual. The constructor cannot be virtual, as it requires information about the accurate type of the object in order to

construct it properly. The virtual destructors are implemented in a similar manner to virtual functions. In constructors and destructors, pecking order (hierarchy) of base and derived classes is constructed. Destructors of derived and base classes are called when a derived class object addressed by the base class pointer is deleted.

For example, a derived class object is constructed using a new operator. The base class pointer object holds the address of the derived object. When the base class pointer is destructed using the delete operator, the destructor of the base and derived class is executed. The following program explains this.

A program to define virtual destructors.

```
#include<iostream.h>
#include<conio.h>
class B
{
public:
B() {cout<<endl<<"In constructor of class B";}
virtual ~B() {cout<<endl<<"In destructor of class B";}
};
class D: public B
{
public:
D() {cout<<endl<<"In constructor of class D";}
~ D() {cout<<endl<<"In destructor of class D";}
};
void main()
{
clrscr();
B *p;
p= new D;
delete p;
}
```

OUTPUT

```
In constructor of class B
In constructor of class D
In destructor of class D
In destructor of class B
```

Explanation: In the above program, the destructor of the base class B is declared as virtual. A dynamic object is created, and the address of the nameless object that is created is assigned to pointer p. The new operator allocates the memory required for data members. When the object goes out of scope, it should be deleted, and the same should be performed by the statement delete p;. When the derived class object is pointed by the base class pointer object, in order to invoke the base class destructor, virtual destructors are

useful.

9.11 Destructor and Virtual Function

When a virtual function is invoked through a non-virtual member function, late binding is performed. When a virtual function is called through the destructor, the redefined function of a similar class is invoked. Consider the following program.

A program to call virtual function using destructors.

```
#include<iostream.h>
#include<conio.h>
class B
{
public:
~ B() {cout<<endl<<" in virtual destructor";}
virtual void joy() {cout<<endl<<"In joy of class B";}
};
class D : public B
{
public :
~ D()
{
B *p;
p=this;
p->joy();
}
void joy() {cout<<endl<<" in joy() of class D";}
};
void main()
{
clrscr();
D X;
}
```

OUTPUT

```
in joy() of class D
in virtual destructor
```

Explanation: In the above program, the destructor of the derived class function joy() is invoked. The member function joy() of the derived class is invoked followed by the virtual destructor.

9.12 Summary

The word poly means many, and morphism means several forms. Both the words are derived from Greek language. Thus, by combining these two words, a new whole word called polymorphism is created, which means various forms. The information pertaining to various overloaded member functions is to be given to the compiler while compiling. This is called early binding or static binding. The deciding function call at run time is called run time or late or dynamic binding. Dynamic binding permits to suspend the decision of choosing a suitable member function until run time. Pointers to the object of a base class are type compatible with pointers to the object of a derived class. The reverse is not possible. Virtual functions of the base class should be redefined in the derived classes. The programmer can define a virtual function in a base class, and can then use a similar function name in any derived class. Addresses of different objects can be stored in an array to invoke the function dynamically. In practical applications, the member function of the base class is rarely used for doing any operation; such functions are called do-nothing functions, dummy functions, or pure virtual functions. All other derived classes without pure virtual functions are called concrete classes. Abstract classes are similar to a skeleton on which new classes are designed to assemble a well-designed class hierarchy. They are not used for object declaration. Virtual functions can be invoked using a pointer or a reference. If an object of a derived class is assigned to a base class object, the compiler allows it. However, it copies only the base class members of the object, and this process is known as object slicing. It is possible to invoke a virtual function using a constructor. The constructor makes the virtual mechanism illegal. We have learned how to declare virtual functions. Likewise, destructors can be declared as virtual. The constructor cannot be virtual. The virtual destructors are implemented in a similar manner to virtual functions. Destructors of derived and base classes are called when a derived class object addressed by the base class pointer is deleted.

9.13 Questions

1. What is a virtual destructor and when would you use one?
2. What are virtual functions? Explain with an example.
3. Differentiate between early binding and late binding.
4. What is an abstract class? Give example of an abstract class definition.
5. How a pure virtual class is defined. Show with an example.
6. Define object slicing.
7. Show with an example where a virtual function is called through constructor.
8. Show with an example that an abstract class can have a constructor.
9. Explain with an example concept of virtual destructor.
10. Write a program where a virtual function show() of base class is redefined in derived class.

9.14 Suggested Readings

1. Object oriented Programming with ANSI and Turbo C++ (Pearson): Ashok N Kamthane
2. Object Oriented Programming with C++, 3/e by E. Balagurusamy, McGraw Hill