Unit

8

Virtual Function and Polymorphism

1. Introduction

We have seen how to create an object and an array of objects. When an object of a class is created, memory is allocated statically for that object. For dynamic memory allocation, we have to use pointers. A pointer can be used to invoke member functions. In the case of inheritance, the derived class inherits properties from the base class. The derived class may have overridden functions from the base class as well as have its own functions. We can use pointers with the base as well as derived classes. In this chapter we will see how to use pointers with base and derived classes and the need for a special type of function called 'virtual function'. Polymorphism is an important property of object oriented programming. We will also see how virtual functions can be used in achieving run time polymorphism.

2. Pointers to Object and Pointer to Derived Classes

We have seen how to create a pointer to a class and create objects dynamically using the new operator.

Syntax:

```
classname * pointer;
```

Example:

```
student * ptr;
```

To create an object dynamically, we use the new operator.

```
ptr = new student;
```

The pointer can be used to call member functions of the object. For example:

```
ptr->accept();
ptr->display();
```

In case of inheritance, a base class pointer can point to either a base class object or a derived class object. However, a derived class pointer cannot point to a base class object.

Example: Consider two classes - base and derived.

```
base b, *bptr;
derived d;
bptr = &b;          // refers to base class object
bptr = &d;          // refers to derived class object
derived *dptr;
dptr = &b;          //invalid
dptr = &d;          //refers to derived class object
```

In the following program, we invoke the member function display() of the base and derived class using base class pointer bptr.

Program: To illustrate pointer to base and derived objects

```
#include<iostream.h>
class base
{
  public:
   void display()
   {   cout<<"In display() of base class"<<endl;   }
};
class derived : public base
{
  public:
  void display()
  { cout<<"In display() of derived class"<<endl;    }
};
```

```
int main()
{
    base *bptr, b;
    derived d ;
    bptr = &b;
    bptr->display();
    bptr=&d;
    bptr->display();
    return 0;
}
```

Output

In display() of base class
In display() of base class

As seen from the output above, the display() function of the base class is invoked twice. Even if the pointer points to the derived class object, the base class function is invoked. Using the base class pointer, we can only access base class members and functions even if the pointer points to the derived class object.

To make the base class pointer invoke the derived class function:

i. Explicit typecasting should be done. This means that we should convert the type of the base class pointer to a derived class pointer as shown below:

   `(derived *)bptr`

   The derived class function can be invoked using the statement:

   `((derived *)bptr)->display();`

ii. Use virtual functions.

Need to access derived class object using base class pointer

Why should we use the base class pointer to access the derived class object? Why not use a derived class pointer instead? There are many *examples* where we may need to use a base class pointer to access derived class members.

To understand this, let us consider the following class hierarchy:

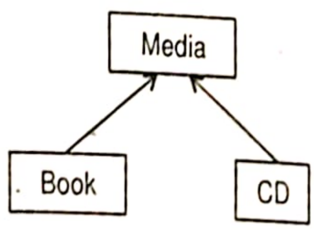


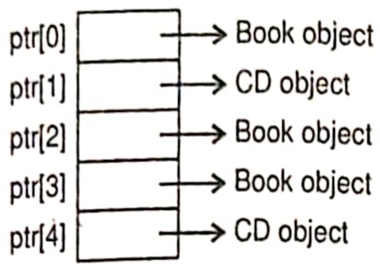
Figure 8.1: Inheritance Example

Assume that all three classes have methods accept() and display(). Let us assume that the user wants to create 5 objects which can either be of type Book or CD. For this purpose, we may want to create an array of objects. But since we don't know exactly how many book and CD objects need t created, we will have to use an array of pointers. What should the array be declared as? Should declared of type Book or CD? It should be of the type Media since Media is the base class of classes.

```
Media * ptr[5];
```

The objects can be created using the new operator depending on whether the user wants to crea Book object or a CD object. Hence, some pointers will point to Book objects while some will p to CD objects

The array may look like this:



If we want to display the details of all objects, the code will be as follows:

```
for(i=0; i<5 ; i++)
   ptr->display();  //calls display() of Media
```

However, this will only call the base class display() function and not the derived class functions. V need to invoke the derived class functions using the base class pointer. Moreover, typecasting w not be possible here because we don't have any information about the type of each object. Hence, order to ensure that the base class pointer invokes the method of the correct derived class, the ba class function should be declared as "**virtual**".

3. Virtual Functions and Pure Virtual Functions

As seen earlier, when the base and derived class have functions with the same name, a pointer to th base class will always invoke the base class function. To ensure that the base class pointer invoke the function of the correct class (either base or derived), the base class function should be declared a virtual.

When we use the same function name in both the base and derived classes, the function in the bas class is declared as *virtual* using the keyword **virtual** preceding its name. When a function is mad

virtual, C++ determines which function to use at run time based on the type of object pointed to by the base pointer, rather than the type of the pointer. Thus, by making the base pointer to point to different objects, we can execute different versions of the **virtual** function. And this choice is made at run time. This is how **run-time polymorphism** is achieved.

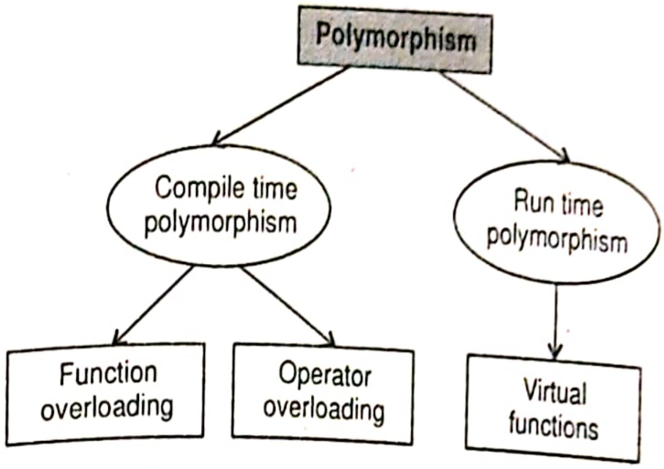


Figure 8.2: Types of polymorphism

To make a member function virtual, the keyword **virtual** is used only in the method declaration but not in the function definition.

The general syntax of the virtual function declaration is:

```
virtual return-type function-name(argument-list);
```

Example:

```
class base
{
  ...
  public:
    virtual void display(); //declaration
};
void base::display()
{
  //code
}
```

Let us now modify *program* so that the base pointer can invoke the base class function if it points to a base class object and invoke a derived class function if it points to a derived class object.

Program : Virtual functions

```
#include<iostream.h>
class base
{
  public:
```

```
  virtual void display()
  {
   cout<<"In base class display "<<endl;
  }
};
class derived : public base
{
  public:
  void display()
  {
   cout<<"In derived class display "<<endl;
  }
};
int main()
{
  base b, *bptr;
  derived d;
  bptr = &b;
  bptr->display(); //calls base class display()
  bptr=&d;
  bptr->display(); //calls derived class display()
  return 0;
}
```

Output

In base class display
In derived class display

3.1 Rules for Virtual Functions

1. Only a member function of a class can be declared as virtual. It is an error to declare a non member function of a class as virtual.
2. The keyword virtual should only be used in function declaration and not repeated in the definition if the definition occurs outside the class declaration.
3. A virtual function cannot be static member function.
4. They are accessed by using object pointers.
5. A virtual function can be a friend of another class.
6. A virtual function in a base class must be defined, even though it may not be used.
7. The prototypes of the base class version of a virtual function and all the derived class versions must be identical. If two functions with the same name have different prototypes, C++ considers them as overloaded functions, and the virtual function mechanism is ignored.
8. Constructors cannot be declared virtual. However, we can have virtual destructors.

9. While a base pointer can point to any derived type, we cannot use a pointer to a derived class to access an object of the base type.

10. When a base pointer points to derived class, incrementing or decrementing it will not make it to point to the next object of the derived class. Therefore, we should not use this method to move the pointer to the next object.

11. If a virtual function is defined in the base class, it need not be necessarily redefined in the derived class. In such cases, calls will invoke the base function.

8.2 Pure Virtual Functions

In the previous section we have seen that when the base class function is declared as virtual and redefined in the derived class, the base class pointer can invoke the base as well as derived class function on the basis of the object type. However, in some cases, we may not want to invoke the base class function because it does not perform any meaningful task. The derived class function performs the corresponding task. In such cases, the base class function can be made a **pure virtual function**. This also ensures that all derived classes must define (override) that function.

The functions which are only declared but not defined in the base class are called *pure virtual functions*. A function is made pure virtual by preceding its declaration with the keyword `virtual` and assigning it value `0`.

The general form of pure virtual function declaration is

```
virtual return-type function-name(argument_list) = 0;
```

Example:

```
virtual void display() = 0;
```

When a virtual function is made pure, any derived class must provide its own definition. If the derived class fails to override the pure virtual function, a compile time error will result.

The following program illustrates how a pure virtual function is defined, declared and invoked from the object of a derived class through the pointer of the base class. Consider the following class hierarchy.

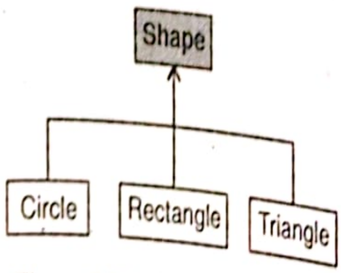


Figure 8.3: Inheritance example

The Shape class is the base class for Circle, Rectangle and Triangle classes. The Shape class has members – dim1, dim2. It has a member function called area(). But since the formula for calculating the area is different for each derived shape, the area() function cannot be defined in the Shape class. Hence it can be declared as a pure virtual function. This function will have to be defined in each of the derived classes.

Program : Pure Virtual Functions

```
#include<iostream.h>
class Shape
{
 protected:
    float dim1, dim2;
 public:
  static float pi;
  Shape(float d1=0, float d2=0)
  {
    dim1=d1; dim2=d2;
  }
  virtual float area()=0; //pure virtual function
};
float Shape::pi = 3.142;
class Circle : public Shape
{
  public:
   Circle(float r):Shape(r)
   { }
   float area()
   {  return pi*radius*radius;
   }
};
class Rectangle : public Shape
{
  public:
   Rectangle(float l, float b):Shape(l,b)
   {     }
   float area()
   {
     return dim1*dim2;
   }
};
```

```
ass Triangle : public Shape
{
public:
    Triangle(float b, float h): Shape(b, h)
    { }
    float area()
    {
        return 0.5*dim1*dim2;
    }
};
nt main()
{
    Shape *ptr;
    ptr = new Circle(2);
    cout<<"Area of Circle = "<<ptr->area()<<endl;
    ptr = new Rectangle(10,20);
    cout<<"Area of Rectangle = "<<ptr->area()<<endl;
    ptr = new Triangle(10,40);
    cout<<"Area of Triangle = "<<ptr->area()<<endl;
    return 0;
}
```

Output

```
Area of Circle = 12.568
Area of Rectangle = 50.271999
Area of Triangle = 125.68
```

3.3 Difference between Virtual and Pure Virtual Functions

1. A virtual function is defined i.e. it has a code. Pure virtual function does not have any definition.
2. A virtual function need not be redefined in the derived class. A pure virtual function has to be defined in the derived class.
3. We can create objects of a class having a virtual function but a class with a pure virtual function is abstract i.e. it cannot be instantiated.
4. A class containing a virtual function need not be inherited. A class containing a pure virtual function must be inherited and the derived class must define the pure virtual function.

3.4. Abstract Classes and Virtual Functions

We have seen the concept of abstract class in the previous chapter. An abstract class is a class which cannot be instantiated i.e. its objects cannot be created. In the previous program, the Shape class is the base class which has three derived classes. As seen in main, only objects of the derived classes are created. The Shape class only serves as a base class and we will not create objects of the Shape class. Such a class is an abstract class. Any class which has a pure virtual function is an abstract class.

4. Run Time Type Information (RTTI)

Run time type information is a mechanism to obtain information about the type of an object during run time. This feature can only be used with classes that are polymorphic i.e. they have atleast one virtual function.

RTTI is a powerful tool which allows the programmer more control and functionality during run time especially to typecast objects.

The RTTI mechanism contains:

i. The dynamic_cast operator

ii. The typeid operator

iii. The type_info structure

4.1 Dynamic Typecasting using dynamic_cast

This operator is used only with pointers and references to objects. Its purpose is to ensure that the result of the type conversion is a valid object of the required class. Dynamic cast is used to convert pointers and references at run-time, for converting a base class pointer to a derived class pointer.

Syntax

```
dynamic_cast <datatype> (expression)
```

Examples,

The following program shows how we can cast a base class pointer to a derived class using the dynamic_cast operator. Here, we cast a pointer to class Animal (base class) a pointer to either the Dog or Bird class which are its base classes.

Program : dynamic_cast operator

```
#include<iostream.h>
class Animal
{
  virtual void run() {}
  virtual void fly() {}
};
class Dog: public Animal
{
  public:
  void run()
  {      cout<<"run"<<endl;      }
};
class Bird : public Animal
{
  public:
   void fly()
   {       cout<<"fly"<<endl;      }
};
void move(Animal *a)
{
  Dog *ptr1 = dynamic_cast<Dog *>(a);
      if(ptr1) { //cast succeeded
         ptr1->run();
} else
   {
    Bird *ptr2 = dynamic_cast<Bird *>(a);
      ptr2->fly();
   }
}
void main()
{
   Dog d1;
   move(&d1);
   Bird b1;
   move(&b1);
}
```

Output

run

fly

4.2 Typeid and type_info

The `typeid` operator is used to determine the class of an object at runtime. It returns a reference to a `std::type_info` object, which describes the "object".

Syntax:

```
typeid (object)
```

The type_info class holds information about a type, including the name of the type and functions to compare objects etc. Its member functions are:

Function	Purpose
operator ==	Compares types
operator !=	Compares types
name	Get type name
hash_code	Get hash code for type. This value is identical for the same types.
before	Checks whether the referred type precedes referred type of another object.

Example

```
class Person
{
  public:
  ...
   virtual ~Person() {}
};
class Employee : public Person
{
   ...
};
int main ()
{
   Person person;
   Employee employee;
   Person *ptr = &employee;
   cout << typeid(person).name() << endl;
   cout << typeid(employee).name() << endl;
   cout << typeid(ptr).name() << endl;
   cout << typeid(*ptr).name() << endl;
}
```

EXERCISES

A. State True or False

1. A pointer to a base class can point to an object of a derived class of that base class.
2. Virtual functions allow us to use the same function call to execute member functions of different classes.
3. A pure virtual function in a class will make the class abstract.
4. An abstract class is never used as a base class.
5. A derived class can never be made an abstract class.
6. RTTI can be used on any class.

B. Review Questions

1. What is a virtual function and what are the advantages of declaring a virtual function in a program?
2. Explain how run time polymorphism is achieved using virtual functions.
3. What is virtual base class and an abstract base class?
4. What are the syntactic rules to be observed while defining the keyword virtual?
5. What is RTTI?
6. Explain dynamic_cast and typeid.

C. Programming Exercises

1. Write class declarations and member function definitions for a C++ base class to represent a Media item (id, description, price). Design derived classes CD (capacity) and book (no of pages). Build a shopping list of 'n' media items where each item can be either Book or CD. Also display the total amount with the details of the item.

Answers

A.

1.	True	2.	True
3.	True	4.	False
5.	False	6.	False

VISION