Object Oriented Programming in C++

Segment-6

Course Code: 2301

Prepared by Sumaiya Deen Muhammad

Lecturer, CSE, IIUC

Contents

Virtual Functions:

- Pointers to derived classes,
- Applying Polymorphism using virtual functions,
- Polymorphic class,
- Pure Virtual functions,
- Abstract classes,
- early binding, and late binding.

Virtual Function

Virtual means existing in appearance but not in reality. When virtual functions are used, a program that appears to be calling a function of one class may in reality be calling a function of a different class.
Virtual functions are used to support run-time polymorphism.
Polymorphism is supported by C++ in two ways.
First, it is supported at compile time, through the use of overloaded operators and functions.
Second, it is supported at run-time, through the use of virtual functions.
The foundation of virtual functions and run-time polymorphism is: pointers to derived classes.
Virtual Function has been discussed later in details.

Pointers to derived classes

A Pointer declared as a Pointer to a base class can also be used to point to any class derived from that base. For example: assume two classes called base and derived, where derived inherits base.

```
base *p; // base class pointer

base base_ob; // object of type base
derived derived_ob; // object of type derived

// p can, of course, point to base objects
p = &base_ob; // p points to base object

// p can also point to derived objects without error
p = &derived_ob; // p points to derived object
```

Pointers to derived classes

Base pointer can point to an object of any class which is derived from that base class. The reverse is not true. That is, a pointer of the derived class cannot be used to access an object of the base class.

Example-1

```
// Demonstrate pointer to derived class.
#include <iostream>
using namespace std;

class base {
  int x;
public:
  void setx(int i) { x = i; }
  int getx() { return x; }
};

class derived : public base {
  int y;
public:
  void sety(int i) { y = i; }
  int gety() { return y; }
};

int main()
{
  base *p; // pointer to base type
  base b_ob; // object of base
  derived d_ob; // object of derived
```

Example-1 (contd.)

```
  // use p to access base object
  p = &b_ob;
  p->setx(10); // access base object
  cout << "Base object x: " << p->getx() << '\n';

  // use p to access derived object
  p = &d_ob; // point to derived object
  p->setx(99); // access derived object

  // can't use p to set y, so do it directly
  d_ob.sety(88);
  cout << "Derived object x: " << p->getx() << '\n';
  cout << "Derived object y: " << d_ob.gety() << '\n';

  return 0;
}
```

Introduction to Virtual Function

A virtual function is a member function that is declared within a base class and redefined by a derived class. To create a virtual function, precede the function's declaration with the keyword **virtual**. When a virtual function is redefined by a derived class, the keyword virtual is not needed.

Example-2

```
class Base //base class
{
public:
virtual void show()   //virtual function
{ cout << "Base\n"; }
};
class Derv1 : public Base //derived class 1
{
public:
void show()
{ cout << "Derv1\n"; }
};
class Derv2 : public Base //derived class 2
{
public:
void show()
{ cout << "Derv2\n"; }
};
```

```
int main()
{
Derv1 dv1;        //object of derived class 1
Derv2 dv2;        //object of derived class 2
Base* ptr;        //pointer to base class
ptr = &dv1;       //put address of dv1 in pointer
ptr->show();      //execute show()

ptr = &dv2;       //put address of dv2 in pointer
ptr->show();      //execute show()
return 0;
}
```

Abstract Class & Pure Virtual Function

- Defines an abstract type which cannot be instantiated, but can be used as a base class.
- A class is made abstract by declaring at least one of its functions as **pure virtual** function.
- A pure virtual function is specified by placing "= 0" in its declaration functions as **pure virtual** function.

Abstract Class & Pure Virtual Function (contd.)

```
class Base //base class
{
public:
virtual void show() = 0; //pure virtual function
};
class Derv1 : public Base //derived class 1
{
public:
void show()
{ cout << “Derv1\n”; }
};
```

```
class Derv2 : public Base
{//derived class 2
public:
void show()
{
        cout << “Derv2\n”; }
};
```

Abstract Class & Pure Virtual Function (contd.)

```
int main()
{
    Base* arr[2]; //array of pointers to base class
    Derv1 dv1; //object of derived class 1
    Derv2 dv2; //object of derived class 2
    arr[0] = &dv1; //put address of dv1 in array
    arr[1] = &dv2; //put address of dv2 in array
    arr[0]->show(); //execute show() in both objects
    arr[1]->show();
    return 0;
}
```

Output:

Derv1

Derv2

Here the virtual function show() is declared as

virtual void show() = 0; // pure virtual function

The equal sign here has nothing to do with assignment; the value 0 is not assigned to anything.

The =0 syntax is simply how we tell the compiler that a virtual function will be pure.

Polymorphic Class

A **class** having at least one virtual function is called a **polymorphic** type.

Applying Polymorphism: Early binding & Late binding

- Polymorphism is important because it can greatly simplify complex systems.
- Early binding refers to those events that can be known at compile time. Specifically it refers to those functions calls that can be resolved during compilation.
- The main advantage of early binding is that it is very efficient.
- Disadvantage: lack of flexibility.

Applying Polymorphism: Early binding & Late binding

- Late Binding refers to events that must occur at run time. A late bound function call is one in which the address of the function to be called is not known until the program runs. In C++, a virtual function is a late bound object.
- Advantage: flexibility at run time.
- Disadvantage: there is more overhead associated with a function call. It makes function calls slower that early binding.