Virtual Functions

Chapter 10

Department of CSE, BUET

Objectives

- Polymorphism in C++
- Pointers to derived classes
- Important point on inheritance
- Introduction to virtual functions
- Virtual destructors
- More about virtual functions
- Final comments
- Applying polymorphism

POLYMORPHISM IN C++

- 2 types
  - Compile time polymorphism
    - Uses static or early binding
    - Example: Function and operator overloading
  - Run time polymorphism
    - Uses dynamic or early binding
    - Example: Virtual functions

Pointers to Derived Classes

- C++ allows base class pointers to point to derived class objects.
- Let we have –
  - class base { … };
  - class derived : public base { … };
- Then we can write –
  - base *p1; derived d_obj; p1 = &d_obj;
  - base *p2 = new derived;

POINTERS TO DERIVED CLASSES (CONTD.)

- Using a base class pointer (pointing to a derived class object) we can access only those members of the derived object **that were inherited from the base**.
  - It is different from the behavior that Java shows.
  - We can get Java-like behavior using virtual functions.
- This is because the **base pointer** has knowledge only of the base class.
- It knows nothing about the members added by the derived class.

POINTERS TO DERIVED CLASSES (CONTD.)

```
class base {
public:
   void show() {
      cout << "base\n";
   }
};
class derived : public base {
public:
   void show() {
      cout << "derived\n";
   }
};
```

```
void main() {
   base b1;
   b1.show(); // base
   derived d1;
   d1.show(); // derived
   base *pb = &b1;
   pb->show(); // base
   pb = &d1;
   pb->show(); // base
}
```

- All the function calls here are statically bound

Department of CSE, BUET

POINTERS TO DERIVED CLASSES (CONTD.)

- While it is permissible for a base class pointer to point to a derived object, the reverse is not true.
  - base b1;
  - derived *pd = &b1; // compiler error
- We can perform a **downcast** with the help of type-casting, but should use it with caution (see next slide).

POINTERS TO DERIVED CLASSES (CONTD.)

- Let we have –
  - class base { … };
  - class derived : public base { … };
  - class xyz { … }; // having no relation with "base" or "derived"
- Then if we write –
  - base b_obj; base *pb; derived d_obj; pb = &d_obj; // ok
  - derived *pd = pb; // compiler error
  - derived *pd = (derived *)pb; // ok, valid downcasting
  - xyz obj; // ok
  - pd = (derived *)&obj; // invalid casting, no compiler error, but may cause run-time error
  - pd = (derived *)&b_obj; // invalid casting, no compiler error, but may cause run-time error

POINTERS TO DERIVED CLASSES (CONTD.)

- In fact using type-casting, we can use pointer of any class to point to an object of any other class.
  - The compiler will not complain.
  - During run-time, the address assignment will also succeed.
  - But if we use the pointer to access any member, then it may cause run-time error.
- Java prevents such problems by throwing "ClassCastException" in case of invalid casting.

Pointers to Derived Classes (contd.)

- Pointer arithmetic is relative to the data type the pointer is declared as pointing to.
- If we point a base pointer to a derived object and then increment the pointer, it will not be pointing to the next derived object.
- It will be pointing to (what it thinks is) the next base object !!!
- **Be careful about this.**

IMPORTANT POINT ON INHERITANCE

- In C++, only public inheritance supports the perfect IS-A relationship.
- In case of private and protected inheritance, we cannot treat a derived class object in the same way as a base class object
  - Public members of the base class becomes private or protected in the derived class and hence cannot be accessed directly by others using derived class objects
- If we use private or protected inheritance, we cannot assign the address of a derived class object to a base class pointer directly.
  - We can use type-casting, but it makes the program logic and structure complicated.
- This is one of the reason for which Java only supports public inheritance.

Introduction to Virtual Functions

- A virtual function is a member function that is declared within a base class and redefined (called ***overriding***) by a derived class.
- It implements the "one interface, multiple methods" philosophy that underlies polymorphism.
- The keyword **virtual** is used to designate a member function as virtual.
- Supports run-time polymorphism with the help of base class pointers.

Introduction to Virtual Functions (Contd.)

- While redefining a virtual function in a derived class, the function signature must match the original function present in the base class.
- So, we call it ***overriding***, not overloading.
- When a virtual function is redefined by a derived class, the keyword **virtual** is not needed (but can be specified if the programmer wants).
- The "virtual"-ity of the member function continues along the inheritance chain.
- A class that contains a virtual function is referred to as a ***polymorphic class***.

Introduction to Virtual Functions (contd.)

- class base {
- public:
-   **virtual** void show() {
-     cout << “base\n”;
-   }
- };
- class derived : **public base** {
- public:
-   void show() {
-     cout << “derived\n”;
-   }
- };

- void main() {
-   base b1;
-   b1.show(); // base - (s.b.)
-   derived d1;
-   d1.show(); // derived – (s.b.)
-   base *pb = &b1;
-   pb->show(); // base - (d.b.)
-   **pb = &d1;**
-   **pb->show(); // derived (d.b.)**
- }
- Here,
  - s.b. = static binding
  - d.b. = dynamic binding

Introduction to Virtual Functions (contd.)

```
class base {
public:
   virtual void show() {
      cout << "base\n";
   }
};
class d1 : public base {
public:
   void show() {
      cout << "derived-1\n";
   }
};
```

```
class d2 : public base {
public:
   void show() {
      cout << "derived-2\n";
   }
};
void main() {
   base *pb; d1 od1; d2 od2;
   int n;
   cin >> n;
   if (n % 2) pb = &od1;
   else pb = &od2;
   pb->show(); // guess what ??
}
```

Run-time polymorphism

VIRTUAL DESTRUCTORS

- Constructors cannot be virtual, but destructors can be virtual.
- It ensures that the derived class destructor is called when a base class pointer is used while deleting a dynamically created derived class object.

Virtual Destructors (contd.)

```
class base {
public:
  ~base() {
    cout <<  "destructing base\n";
  }
};
class derived : public base {
public:
  ~derived() {
    cout << "destructing derived\n";
  }
};
```

```
void main() {
  base *p = new derived;
  delete p;
}
```

- Output:
  - destructing base

Using non-virtual destructor

Department of CSE, BUET

Virtual Destructors (contd.)

```
class base {
public:
  virtual ~base() {
    cout <<  "destructing base\n";
  }
};
class derived : public base {
public:
  ~derived() {
    cout << "destructing derived\n";
  }
};
```

```
void main() {
  base *p = new derived;
  delete p;
}
```

- Output:
  - destructing derived
  - destructing base

Using virtual destructor

More About Virtual Functions

- If we want to omit the body of a virtual function in a base class, we can use pure virtual functions.
  - virtual ret-type func-name(param-list) = 0;
- It makes a class an ***abstract class***.
  - We cannot create any objects of such classes.
- It forces derived classes to override it.
  - Otherwise they become abstract too.

More About Virtual Functions (contd.)

- Pure virtual function
  - Helps to guarantee that a derived class will provide its own redefinition.
- We can still create a pointer to an abstract class
  - Because it is at the heart of run-time polymorphism
- When a virtual function is inherited, so is its virtual nature.
- We can continue to override virtual functions along the inheritance hierarchy.

Final Comments

- Run-time polymorphism is not automatically activated in C++.
- We have to use virtual functions and base class pointers to enforce and activate run-time polymorphism in C++.
- But, in Java, run-time polymorphism is automatically present as all **non-static methods** of a class are by default virtual in nature.
  - We just need to use superclass references to point to subclass objects to achieve run-time polymorphism in Java.

APPLYING POLYMORPHISM

- Early binding
  - Normal functions, overloaded functions
  - Nonvirtual member and friend functions
  - Resolved at compile time
  - Very efficient
  - But lacks flexibility
- Late binding
  - Virtual functions accessed via a base class pointer
  - Resolved at run-time
  - Quite flexible during run-time
  - But has run-time overhead; slows down program execution

LECTURE CONTENTS

- Teach Yourself C++
  - Chapter 10 (Full, with exercises)
  - Study the examples from the book carefully