Object Oriented Programming with C++ Day 4

Compiled by
M S Anand

Department of Computer Science

Introduction

Text Book(s):

1. “C++ Primer”, Stanley Lippman, Josee Lajoie, Barbara E Moo, Addison-Wesley Professional, 5th Edition..

Inheritance

The capability of a class to derive properties and characteristics from another class is called **Inheritance**. Inheritance is one of the most important features of Object-Oriented Programming.

Inheritance is a feature or a process in which, new classes are created from the existing classes. The new class created is called “derived class” or “child class” and the existing class is known as the “base class” or “parent class”. The derived class now is said to be inherited from the base class.

When we say derived class inherits the base class, it means, the derived class inherits all the properties of the base class, without changing the properties of base class and may add new features to its own. These new features in the derived class will not affect the base class.

Inheritance

Implementing inheritance in C++: For creating a sub-class (derived class)that is inherited from the base class we have to follow the below syntax.

Syntax:
class <derived_class_name> : <access-specifier> <base_class_name> {
//body }Where
class — keyword to create a new class
derived_class_name — name of the new class, which will inherit the base class
access-specifier — either of private, public or protected. If neither is specified, PRIVATE is taken as default
base-class-name — name of the base class

Note: A derived class doesn't inherit ***access*** to private data members. However, it does inherit a full parent object, which contains any private members which that class declares.

Inheritance

Example:

```
1. class ABC : private XYZ          //private derivation
        {           }
2. class ABC : public XYZ           //public derivation
        {           }
3. class ABC : protected XYZ        //protected derivation
        {           }
4. class ABC: XYZ                   //private derivation by default
{       }
```

Inheritance

Note:

1. When a base class is privately inherited by the derived class, public members of the base class become the private members of the derived class and therefore, the public members of the base class can only be accessed by the member functions of the derived class. They are inaccessible to the objects of the derived class.
2. On the other hand, when the base class is publicly inherited by the derived class, public members of the base class also become the public members of the derived class. Therefore, the public members of the base class are accessible by the objects of the derived class as well as by the member functions of the derived class.

Sample programs: Inheritance1.cpp, Inheritance2.cpp
Inheritance3.cpp Inheritance4.cpp

Inheritance

Modes of Inheritance: There are 3 modes of inheritance.

Public Mode:
If we derive a subclass from a public base class, then the public member of the base class will become public in the derived class and protected members of the base class will become protected in the derived class.

Protected Mode:
If we derive a subclass from a Protected base class,then both public members and protected members of the base class will become protected in the derived class.

Private Mode:
If we derive a subclass from a Private base class, then both public members and protected members of the base class will become Private in the derived class.

Program showing different modes of inheritance:
Inheritance5.cpp (Not to be executed)

The table below summarizes the above three modes and shows the access specifier of the members of the base class in the subclass when derived in public, protected and private modes:

Base class member access specifier	Type of Inheritence		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)

Constructors and Destructors in Inheritance

Parent class constructors and destructors are accessible to the child class; hence when we create an object for the child class, constructors and destructors of both parent and child class get executed.

Sample program: CandDInheritance1.cpp

Inheritance in Parametrized Constructor/ Destructor

In the case of the default constructor, it is implicitly accessible from parent to the child class; but parameterized constructors are not accessible to the derived class automatically; for this reason, an explicit call has to be made in the child class constructor to access the parameterized constructor of the parent class to the child class using the following syntax

<class_name>:: constructor(arguments)

Sample program: CandDInheritance2.cpp

Inheritance

When an object of a derived class is created, the base class' constructor will be called first, followed by the derived class' constructor. When a derived object is destroyed, its destructor is called first, followed by the base class' destructor.

Put differently, constructors are executed in their order of derivation. Destructors are executed in reverse order of derivation.

Inheritance

Function overriding

Function overriding in C++ is termed as the redefinition of base class function in its derived class with the same signature i.e. return type and parameters. It falls under the category of Runtime Polymorphism.

A simple example: FunctionOverride1.cpp

Variations in Function overriding

1. Call Overridden Function From Derived Class

A simple example: FunctionOverride2.cpp

2. **Call Overridden Function Using Pointer**

An example: FunctionOverride3.cpp

3. Access of Overridden Function to the Base Class

An example: FunctionOverride4.cpp

4. Access to Overridden Function

An example: FunctionOverride5.cpp

Virtual functions

A virtual function is a member function in the base class that we expect to redefine in derived classes.

Basically, a virtual function is used in the base class in order to ensure that the function is **overridden**.

This especially applies to cases where a pointer of base class points to an object of a derived class.

***Note**: In C++ what calling a virtual functions means is that; if we call a member function then it could cause a different function to be executed instead depending on what type of object invoked it. Because overriding from derived classes hasn't happened yet, the virtual call mechanism is disallowed in constructors. Also to mention that objects are built from the ground up or follows a bottom to top approach.*

Inheritance

Sample program without virtual function:

WithoutVirtualFunction.cpp

We store the address of each child's class **Rectangle** and **Square** object in **s** and then we call the **get_Area()** function on it.

Ideally, it should have called the respective **get_Area()** functions of the child classes but instead, it calls the get_Area() defined in the base class.

This happens due to static linkage which means the call to get **get_Area()** is getting set only once by the compiler which is in the base class.

Inheritance

Sample program with virtual function:
WithVirtualFunction.cpp

What is the use?
Virtual functions allow us to create a list of base class pointers and call methods of any of the derived classes without even knowing the kind of derived class object.

A real life example: RealLife.cpp (not to be compiled)

Sample program: VirtualFunction1.cpp

Runtime polymorphism is achieved only through a pointer (or reference) of the base class type. Also, a base class pointer can point to the objects of the base class as well as to the objects of the derived class. In the above code, the base class pointer ‘bptr’ contains the address of object ‘d’ of the derived class.

Late binding (Runtime) is done in accordance with the content of the pointer (i.e. location pointed to by pointer) and Early binding (Compile-time) is done according to the type of pointer since the print() function is declared with the virtual keyword so it will be bound at runtime (output is *print derived class* as the pointer is pointing to object of derived class) and show() is non-virtual so it will be bound during compile time (output is *show base class* as the pointer is of base type).

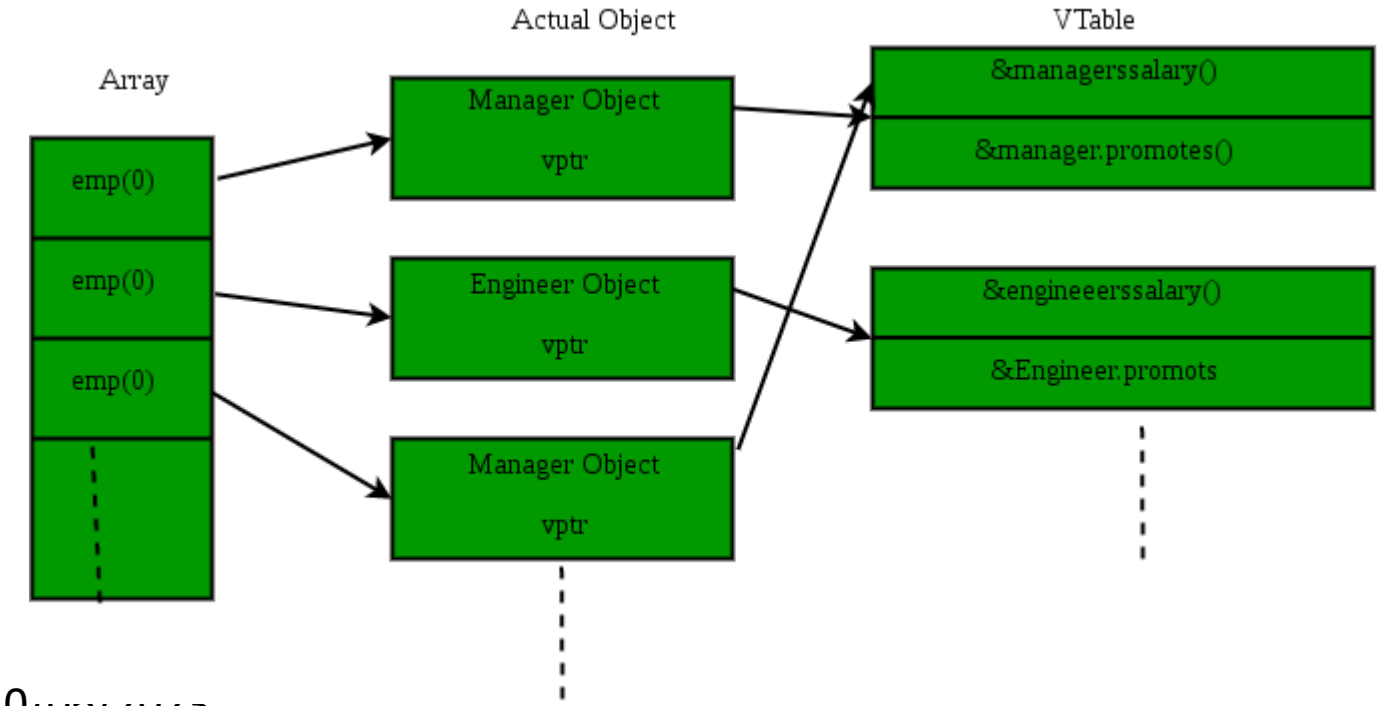
Sample program: VirtualFunction2.cpp

Rules for Virtual Functions

1. Virtual functions cannot be static.
2. A virtual function can be a friend function of another class.
3. Virtual functions should be accessed using a pointer or reference of base class type to achieve runtime polymorphism.
4. The prototype of virtual functions should be the same in the base as well as the derived class.
5. They are always defined in the base class and overridden in a derived class. It is not mandatory for the derived class to override (or re-define the virtual function), in that case, the base class version of the function is used.
6. A class may have a virtual destructor but it cannot have a virtual constructor.

How does the compiler perform runtime resolution?

The compiler maintains two data structures to serve this purpose**:**
vtable: A table of function pointers, **maintained per class**.
vptr: A pointer to vtable, maintained **per object instance**.



Inheritance

The compiler adds additional code at two places to maintain and use *vptr*.

1. Code in every constructor. This code sets the *vptr* of the object being created. This code sets *vptr* to point to the *vtable* of the class.

2. Code with polymorphic function call (e.g. *bp->show()* in above code). Wherever a polymorphic call is made, the compiler inserts code to first look for *vptr* using a base class pointer or reference (In the above example, since the pointed or referred object is of a derived type, *vptr* of a derived class is accessed). Once *vptr* is fetched, *vtable* of derived class can be accessed. Using *vtable*, the address of the derived class function *show()* is accessed and called.

Working of Virtual Functions (concept of VTABLE and VPTR)
As discussed in the previous slide, if a class contains a virtual function then the compiler itself does two things.

If an object of that class is created then a **virtual pointer (VPTR)** is inserted as a data member of the class to point to the VTABLE of that class. For each new object created, a new virtual pointer is inserted as a data member of that class.

Irrespective of whether the object is created or not, the class contains as a member **a static array of function pointers called VTABLE**. Cells of this table store the address of each virtual function contained in that class.

Pure virtual functions

Sometimes implementation of all functions cannot be provided in a base class because we don't know the implementation. Such a class is called an **abstract class**. For example, let Shape be a base class. We cannot provide the implementation of function draw() in Shape, but we know every derived class must have an implementation of draw(). Similarly, an Animal class doesn't have the implementation of move() (assuming that all animals move), but all animals must know how to move. We cannot create objects of abstract classes.

A pure virtual function is a function that must be overridden in a derived class and need not be defined in the base class. A virtual function is declared to be "pure" using the curious =0 syntax., But we must override that function in the derived class, otherwise, the derived class will also become an abstract class.

An example of a pure virtual function

```
// An abstract class
class Test
{
        // Data members of class
public:
        // Pure Virtual Function (This is the way to declare a PVF)
        virtual void show() = 0;

        /* Other members */
};
```

A pure virtual function is implemented by classes that are derived from an Abstract class.
An example: PVF1.cpp

Abstract class
A class is abstract if it has at least one pure virtual function**.**

An example: AbstractClass1.cpp

We can have pointers and references of abstract class type**.**
An example: AbstractClass2.cpp

If we do not override the pure virtual function in the derived class, then the derived class also becomes an abstract class.
An example: AbstractClass3.cpp

An abstract class can have constructors**.**
An example: AbstractClass4.cpp

Virtual Destructor

Deleting a derived class object using a pointer of base class type that has a non-virtual destructor results in undefined behavior. To correct this situation, the base class should be defined with a virtual destructor.

For example, the following program results in undefined behavior.

The program: VD1.cpp

Making base class destructor virtual guarantees that the object of derived class is destructed properly, i.e., both base class and derived class destructors are called.

The program: VD2.cpp

As a guideline, any time you have a virtual function in a class, you should immediately add a virtual destructor (even if it does nothing). This way, you ensure against any surprises later.

THANK YOU

M S Anand
Department of Computer Science Engineering

anandms@yahoo.com