Computer Programming
Polymorphism

11th week lecture

엄현상(Eom, Hyeonsang)
School of Computer Science and Engineering
Seoul National University

©COPYRIGHTS 2017 EOM, HYEONSANG ALL RIGHTS RESERVED

Outline

- Polymorphism
- Virtual Functions
- Abstract Classes
- Virtual Destructors
- Q&A

©COPYRIGHTS 2017 EOM, HYEONSANG ALL RIGHTS RESERVED

Animal Hierarchy

- Animal base class
  - Every derived class has function move
- Different animal objects maintained as a vector of Animal pointers
- Program issues same message (move) to each animal generically
- Proper function gets called
  - A Fish will move by swimming
  - A Frog will move by jumping
  - A Bird will move by flying

Polymorphism

- “Program in the general” rather than “program in the specific”
- Process objects of classes that are part of the same class hierarchy as if they are all objects of the hierarchy’s base class
- Occurs when a program invokes a virtual function through a base-class pointer or reference
  - Dynamically chooses the correct function
- new types of objects that can respond to existing messages can be incorporated into such a system without modifying the base system

Invoking Base-Class Functions from Derived-Class Objects

- Base-class pointer to base-class object
  - Invoke base-class functionality
- Derived-class pointer to derived-class object
  - Invoke derived-class functionality

Base-class Pointer to Derived-class Object

- Derived-class object
  - an object of base class
- Base-class functionality
  - Invoked functionality depends on type of the handle used to invoke the function, not type of the object to which the handle points
  - Handle: a reference to an object or a pointer to an object
- Virtual functions
  - Invoke the object type's functionality, rather than invoke the handle type's functionality
  - Crucial to polymorphic behavior

Base-class Pointer to Derived-class Object Cont'd

- Calling functions that exist in base class causes base-class functionality to be invoked
- Calling functions that do not exist in base class (may exist in derived class)
  - Error
  - Derived-class members cannot be accessed from base-class pointers
    - Can be accomplished using downcasting

A Derived-class Pointer to a Base-class Object

- C++ compiler generates error
  - CommissionEmployee (base-class object) is not a BasePlusCommissionEmployee (derived-class object)

```
// Error:
   basePlusCommissionEmployeePtr = &commissionEmployee;
```

- If this were to be allowed, programmer could then attempt to access derived-class members which do not exist
  - Could modify memory being used for other data

Class CommissionEmployee

```
class CommissionEmployee
{
public:
   CommissionEmployee( const string &, const string &, const string &,
    double = 0.0, double = 0.0 );
…
   double earnings() const;
   void print() const;
…
}

class BasePlusCommissionEmployee : public CommissionEmployee
{
public:
   BasePlusCommissionEmployee( const string &, const string &,
      const string &, double = 0.0, double = 0.0, double = 0.0 );

   double earnings() const;
   void print() const;
}
```

Class CommissionEmployee

```
double
    CommissionEmployee::earnings()
    const
{
   return getCommissionRate() *
    getGrossSales();
}

void CommissionEmployee::print()
    const
{
   cout << "commission employee:
    "
      << getFirstName() << ' '
    << getLastName()
      << "\nsocial security
    number: "
    << getSocialSecurityNumber()
      << "\ngross sales: "
    << getGrossSales()
      << "\ncommission rate: "
    << getCommissionRate();
}
```

```
double
    BasePlusCommissionEmployee::e
    arnings() const
{
   return getBaseSalary() +
    CommissionEmployee::earnings(
    );
}

void
    BasePlusCommissionEmployee::p
    rint() const
{
   cout << "base-salaried ";

   CommissionEmployee::print();

   cout << "\nbase salary: "
    << getBaseSalary();
}
```

Class CommissionEmployee

```
int main()
{
   CommissionEmployee
    commissionEmployee(
      "Sue", "Jones", "222-22-
    2222", 10000, .06 );

   CommissionEmployee
    *commissionEmployeePtr = 0;

   BasePlusCommissionEmployee
    basePlusCommissionEmployee(
      "Bob", "Lewis", "333-33-
    3333", 5000, .04, 300 );

   BasePlusCommissionEmployee
    *basePlusCommissionEmployee
    Ptr = 0;
```

```
cout << fixed
 << setprecision( 2 );

cout << "Print base-class
 and derived-class
 objects:\n\n";
commissionEmployee.print();
cout << "\n\n";

 basePlusCommissionEmployee.
 print();

commissionEmployeePtr =
 &commissionEmployee;

cout << "\n\n\nCalling print
 with base-class pointer to
 "
   << "\nbase-class object
 invokes base-class print
 function:\n\n";
```

Class CommissionEmployee

```
commissionEmployeePtr->print();

basePlusCommissionEmployeePtr = &basePlusCommissionEmployee;
cout << "\n\n\nCalling print with derived-class pointer to "
   << "\nderived-class object invokes derived-class "
   << "print function:\n\n";

basePlusCommissionEmployeePtr->print();
```

Handle

```
commissionEmployeePtr = &basePlusCommissionEmployee;
```

Object

```
cout << "\n\n\nCalling print with base-class pointer to "
   << "derived-class object\ninvokes base-class print "
   << "function on that derived-class object:\n\n";
commissionEmployeePtr->print();

cout << endl;

return 0;
}
```

Class CommissionEmployee

…

```
// invoke base-class member functions on derived-class
// object through base-class pointer
string firstName = commissionEmployeePtr->getFirstName();
string lastName = commissionEmployeePtr->getLastName();
string ssn = commissionEmployeePtr-
 >getSocialSecurityNumber();
double grossSales = commissionEmployeePtr->getGrossSales();
double commissionRate = commissionEmployeePtr-
 >getCommissionRate();


// attempt to invoke derived-class-only member functions
// on derived-class object through base-class pointer
double baseSalary = commissionEmployeePtr->getBaseSalary();
commissionEmployeePtr->setBaseSalary( 500 );
```

…

Virtual Functions

- Which class's function to invoke
  - Normally
    - Handle determines which class's functionality to invoke
  - With virtual functions
    - Type of the object being pointed to, not type of the handle, determines which version of a virtual function to invoke
    - Allows program to dynamically (at runtime rather than compile time) determine which function to use
      - Called dynamic binding or late binding

Virtual Functions Cont'd

- Declared by preceding the function's prototype with the keyword virtual in base class
- Derived classes override the function as appropriate
- Once declared virtual, a function remains virtual all the way down the hierarchy
  - Even if that function is not explicitly declared virtual when a class overrides it

Virtual Functions Cont'd

- Once declared virtual, a function remains virtual all the way down the hierarchy Cont'd
  - When a derived class chooses not to override a virtual function from its base class, the derived class simply inherits its base class's virtual function implementation
- Static binding
  - When calling a virtual function using specific object with dot operator, function invocation resolved at compile time
- Dynamic binding
  - Dynamic binding occurs only off pointer and reference handles

Virtual Functions Cont'd

```
class CommissionEmployee
{
public:
   CommissionEmployee( const string &, const string &, const string &,
      double = 0.0, double = 0.0 );
…
   virtual double earnings() const;
   virtual void print() const;
…
}
class BasePlusCommissionEmployee : public CommissionEmployee
{
public:
   BasePlusCommissionEmployee( const string &, const string &,
      const string &, double = 0.0, double = 0.0, double = 0.0 );
…
   virtual double earnings() const;
   virtual void print() const;
…
}
```

Implementation of Earnings and Print

```
double
    CommissionEmployee::earnings()
    const
{
   return getCommissionRate() *
    getGrossSales();
}

void CommissionEmployee::print()
    const
{
   cout << "commission employee:
    "
      << getFirstName() << ' '
    << getLastName()
      << "\nsocial security
    number: "
    << getSocialSecurityNumber()
      << "\ngross sales: "
    << getGrossSales()
      << "\ncommission rate: "
    << getCommissionRate();
}
```

```
double
    BasePlusCommissionEmployee::e
    arnings() const
{
   return getBaseSalary() +
    CommissionEmployee::earnings(
    );
}

void
    BasePlusCommissionEmployee::p
    rint() const
{
   cout << "base-salaried ";

   CommissionEmployee::print();

   cout << "\nbase salary: "
    << getBaseSalary();
}
```

Implementation of Earnings and Print Cont’d

```
int main()
{
   CommissionEmployee
    commissionEmployee(
      "Sue", "Jones", "222-22-
    2222", 10000, .06 );

   CommissionEmployee
    *commissionEmployeePtr = 0;

   BasePlusCommissionEmployee
    basePlusCommissionEmployee(
      "Bob", "Lewis", "333-33-
    3333", 5000, .04, 300 );

   BasePlusCommissionEmployee
    *basePlusCommissionEmployeePt
    r = 0;
```

```
cout << fixed
 << setprecision( 2 );

cout << "Invoking print
 function on base-class and
 derived-class "
   << "\nobjects with static
 binding\n\n";

commissionEmployee.print(); //
 static binding
cout << "\n\n";

 basePlusCommissionEmployee.pr
 int(); // static binding

  cout << "\n\n\nInvoking
 print function on base-class
 and "
   << "derived-class \nobjects
 with dynamic binding";
```

Implementation of Earnings and Print Cont'd

```
commissionEmployeePtr =
 &commissionEmployee;
cout << "\n\nCalling virtual
 function print with base-
 class pointer"
   << "\nto base-class object
 invokes base-class "
   << "print function:\n\n";
commissionEmployeePtr->print();
 // invokes base-class print

basePlusCommissionEmployeePtr
 = &basePlusCommissionEmployee;
cout << "\n\nCalling virtual
 function print with derived-
 class "
   << "pointer\nto derived-
 class object invokes derived-
 class "
   << "print function:\n\n";
basePlusCommissionEmployeePtr-
 >print(); // invokes derived-
 class print

commissionEmployeePtr =
 &basePlusCommissionEmployee;
cout << "\n\nCalling virtual
 function print with base-
 class pointer"
   << "\nto derived-class
 object invokes derived-class
 "
   << "print function:\n\n";

commissionEmployeePtr->print();
cout << endl;

return 0;
}
```

Switch Statements

- switch statement could be used to determine the type of an object at runtime
  - Include a type field as a data member in the base class
  - Enables programmer to invoke appropriate action for a particular object
  - Causes problems
    - A type test may be forgotten
    - May forget to add new types

Switch Statements Cont'd

- By using the C++ polymorphism mechanism to perform the equivalent logic of switch statements, programmers can avoid the kinds of errors typically associated with switch logic
- Consequence of using polymorphism
  - Programs take on a simplified appearance
    - Less branching logic and more simple, sequential code
  - Facilitates testing, debugging, and maintenance

Abstract Classes

- No objects instantiated
  - Incomplete-derived classes must define the missing pieces
  - Too generic to define real objects
- Normally used as base classes, called abstract base classes
  - Provides an appropriate base class from which other classes can inherit
- Concrete classes
  - Classes used to instantiate objects are called
  - Must provide implementation for every member function they define

Abstract Classes Cont'd

- Pure virtual functions
  - A class is made abstract by declaring one or more of its virtual functions to be pure
    - Placing "= 0" (pure specifier) in its declaration

```
virtual void draw() const = 0;
```

  - No implementations
    - Every concrete derived class provide concrete implementations by overriding them
    - If not overridden, derived-class will also be abstract
  - Used when it does not make sense for base class to have an implementation of a function, but the programmer wants all concrete derived classes to implement the function

Abstract Classes Cont'd

- An abstract class defines a common public interface for the various classes in a class hierarchy
- An abstract class contains one or more pure virtual functions that concrete derived classes must override
- An abstract class has at least one pure virtual function
- An abstract class also can have data members and concrete functions (including constructors and destructors)

Abstract Classes Cont'd

- Use the abstract base class to declare pointers and references
  - Can refer to objects of any concrete class derived from the abstract class
  - To manipulate derived-class objects polymorphically
- Polymorphism particularly effective for implementing layered software systems
  - Reading or writing data from and to devices in OS
- Iterator class
  - Can traverse all the objects in a container

Implementation Inheritance vs Interface Inheritance

- Hierarchies designed for implementation inheritance tend to have their functionality high in the hierarchy
  - Each new derived class inherits one or more member functions that were defined in a base class, and the derived class uses the base-class definitions

Implementation Inheritance vs Interface Inheritance Cont'd

- Hierarchies designed for interface inheritance tend to have their functionality lower in the hierarchy
  - A base class specifies one or more functions that should be defined for each class in the hierarchy (i.e., they have the same prototype), but the individual derived classes provide their own implementations of the function(s)

Virtual Functions and Dynamic Binding

- Three levels of pointers ("triple indirection")
- Virtual function table (vtable) created when C++ compiles a class that has one or more virtual functions
  - Contains function pointers (first level of pointers) to virtual functions
  - Used to select the proper virtual function
  - For pure virtual functions, pointers are set to 0

Virtual Functions and Dynamic Binding Cont'd

- Virtual function table (vtable) created when C++ compiles a class that has one or more virtual functions Cont'd
  - Any class that has one or more null pointers in its vtable is an abstract class
- If a non-pure virtual function were not overridden by a derived class
  - The function pointer in the vtable for that class would point to the implemented virtual function up in the hierarchy

Virtual Functions and Dynamic Binding Cont'd

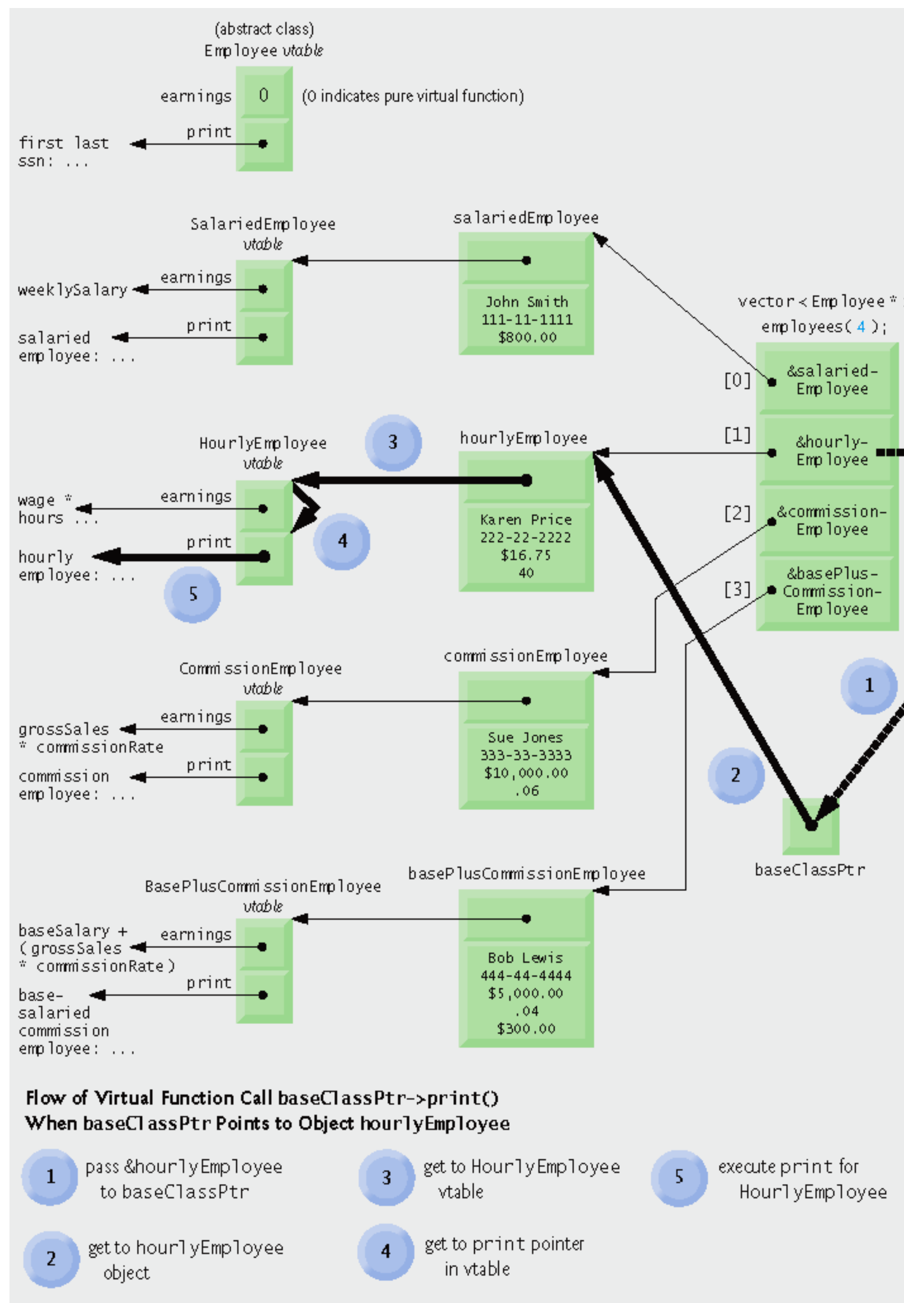
- Second level of pointers
  - Whenever an object of a class with one or more virtual functions is instantiated, the compiler attaches to the object a pointer to the vtable for that class
- Third level of pointers
  - Handles to the objects that receive the virtual function calls

How a Typical Virtual Function Call Executes

- Compiler determines if call is being made via a base-class pointer and that the function is virtual
- Locates entry in vtable using offset or displacement
- Compiler generates code that performs following operations:
  - Select the pointer being used in the function call from the third level of pointers

How a Typical Virtual Function Call Executes Cont'd

- Compiler generates code that performs following operations Cont'd:
  - Dereference that pointer to retrieve underlying object
  - Dereference object's vtable pointer to get to vtable
  - Skip the offset to select the correct function pointer
  - Dereference the function pointer to form the "name" of the actual function to execute, and use the function call operator to execute the appropriate function



C++ How to Program 6th Ed., P. Deitel and H. M. Deitel, Pearson Education, 2008

Dynamic Binding & S/W Distribution

- Dynamic binding enables independent software vendors (ISVs) to distribute software without revealing proprietary secrets
- Software distributions can consist of only header files and object files
  - Software developers can then use inheritance to derive new classes from those provided by the ISVs
  - Other software that worked with the classes the ISVs provided will still work with the derived classes and will use the overridden virtual functions provided in these classes

Downcasting, dynamic_cast, typeid and type_info

- Want to reward BasePlusCommissionEmployees by adding 10% to their base salaries
- Must use run-time type information (RTTI) and dynamic casting to program in the specific
  - Some compilers require that RTTI be enabled before it can be used in a program
    - Consult compiler documentation

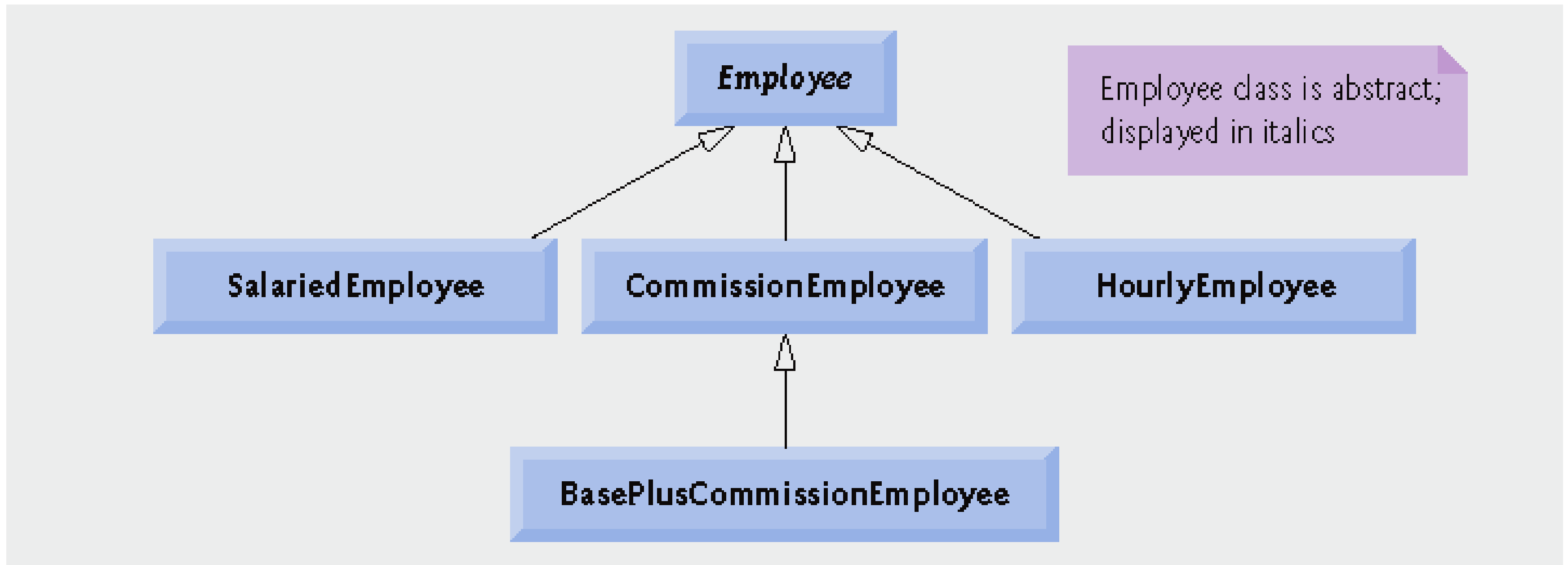
Downcasting, dynamic_cast, typeid and type_info Cont'd

- dynamic_cast operator
  - Downcast operation
    - Converts from a base-class pointer to a derived-class pointer
  - If underlying object is of derived type, cast is performed
    - Otherwise, 0 is assigned
  - If dynamic_cast is not used and attempt is made to assign a base-class pointer to a derived-class pointer
    - A compilation error will occur

Downcasting, dynamic_cast, typeid and type_info Cont'd

- typeid operator
  - Returns a reference to an object of class type_info
    - Contains the information about the type of its operand
    - type_info member function name
      - Returns a pointer-based string that contains the type name of the argument passed to typeid
  - Must include header file <typeinfo>

dynamic_cast



C++ How to Program 6th Ed., P. Deitel and H. M. Deitel, Pearson Education, 2008

Downcasting and dynamic_cast

```
…
#include <typeinfo>
…
int main()
{
   set floating-point output
    formatting
   cout << fixed
    << setprecision( 2 );

   vector < Employee * >
    employees( 4 );
employees[ 0 ] = new
    SalariedEmployee(
      "John", "Smith", "111-11-
    1111", 800 );

   employees[ 1 ] = new
    HourlyEmployee(
      "Karen", "Price", "222-22-
    2222", 16.75, 40 );
```

```
employees[ 2 ] = new
 CommissionEmployee(
   "Sue", "Jones", "333-33-
 3333", 10000, .06 );

employees[ 3 ] = new
 BasePlusCommissionEmployee(
   "Bob", "Lewis", "444-44-
 4444", 5000, .04, 300 );

for ( size_t i = 0; i
 < employees.size(); i++ )
{
   employees[ i ]->print();
   cout << endl;

   BasePlusCommissionEmployee
 *derivedPtr =
      dynamic_cast
 < BasePlusCommissionEmployee
 * > ( employees[ i ] );
```

Downcasting and dynamic_cast Cont’d

```
   if ( derivedPtr != 0 )
   {
      double oldBaseSalary
= derivedPtr-
>getBaseSalary();
      cout << "old base
salary: $"
<< oldBaseSalary << endl;
      derivedPtr-
>setBaseSalary( 1.10 *
oldBaseSalary );
      cout << "new base
salary with 10% increase
is: $"
         << derivedPtr-
>getBaseSalary() << endl;
   }

   cout << "earned $"
<< employees[ i ]-
>earnings() << "\n\n";
}
```

```
for ( size_t j = 0; j
 < employees.size(); j++ )
{
   cout << "deleting
 object of "

 << typeid( *employees[ j ]
 ).name() << endl;

   delete employees[ j ];
}
return 0;
}
```

Virtual Destructors

- Nonvirtual destructors
  - Destructors that are not declared with keyword virtual
  - If a derived-class object is destroyed explicitly by applying the delete operator to a base-class pointer to the object, the behavior is undefined

Virtual Destructors Cont'd

- virtual destructors
  - Declared with keyword virtual
    - All derived-class destructors are virtual
  - If a derived-class object is destroyed explicitly by applying the delete operator to a base-class pointer to the object, the appropriate derived-class destructor is called
    - Appropriate base-class destructor(s) will execute afterwards