13

Object-Oriented Programming: Polymorphism

One Ring to rule them all, One Ring to find them,
One Ring to bring them all and in the darkness bind them.

— John Ronald Reuel Tolkien

The silence often of pure innocence
Persuades when speaking fails.

— William Shakespeare

General propositions do not decide concrete cases.

— Oliver Wendell Holmes

A philosopher of imposing stature doesn't think in a vacuum. Even his most abstract ideas are, to some extent, conditioned by what is or is not known in the time when he lives.

— Alfred North Whitehead

OBJECTIVES

In this chapter you will learn:

- What polymorphism is, how it makes programming more convenient, and how it makes systems more extensible and maintainable.
- To declare and use `virtual` functions to effect polymorphism.
- The distinction between abstract and concrete classes.
- To declare pure `virtual` functions to create abstract classes.
- How to use run-time type information (RTTI) with downcasting, `dynamic_cast`, `typeid` and `type_info`.
- How C++ implements `virtual` functions and dynamic binding "under the hood."
- How to use `virtual` destructors to ensure that all appropriate destructors run on an object.

Outline

13.1 Introduction
13.2 Polymorphism Examples
13.3 Relationships Among Objects in an Inheritance Hierarchy
- 13.3.1 Invoking Base-Class Functions from Derived-Class Objects
- 13.3.2 Aiming Derived-Class Pointers at Base-Class Objects
- 13.3.3 Derived-Class Member-Function Calls via Base-Class Pointers
- 13.3.4 Virtual Functions
- 13.3.5 Summary of the Allowed Assignments Between Base-Class and Derived-Class Objects and Pointers

13.4 Type Fields and `switch` Statements
13.5 Abstract Classes and Pure `virtual` Functions

Outline

13.6 Case Study: Payroll System Using Polymorphism

13.6.1 Creating Abstract Base Class `Employee`

13.6.2 Creating Concrete Derived Class `SalariedEmployee`

13.6.3 Creating Concrete Derived Class `HourlyEmployee`

13.6.4 Creating Concrete Derived Class `CommissionEmployee`

13.6.5 Creating Indirect Concrete Derived Class `BasePlusCommissionEmployee`

13.6.6 Demonstrating Polymorphic Processing

13.7 (Optional) Polymorphism, Virtual Functions and Dynamic Binding "Under the Hood"

13.8 Case Study: Payroll System Using Polymorphism and Run-Time Type Information with Downcasting, `dynamic_cast`, `typeid` and `type_info`

13.9 Virtual Destructors

13.10 (Optional) Software Engineering Case Study: Incorporating Inheritance into the ATM System

13.11 Wrap-Up

13.1 Introduction

- **Polymorphism with inheritance hierarchies**
  - **“Program in the general” vs. “program in the specific”**
  - **Process objects of classes that are part of the same hierarchy as if they are all objects of the base class**
  - **Each object performs the correct tasks for that object’s type**
    - **Different actions occur depending on the type of object**
  - **New classes can be added with little or not modification to existing code**

13.1 Introduction (Cont.)

- Example: `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

13.2 Polymorphism Examples

- Polymorphism occurs when a program invokes a virtual function through a base-class pointer or reference
  - C++ dynamically chooses the correct function for the class from which the object was instantiated
- Example: `SpaceObject`s
  - Video game manipulates objects of types that inherit from `SpaceObject`, which contains member function `draw`
  - Function `draw` implemented differently for the different classes
  - Screen-manager program maintains a container of `SpaceObject` pointers
  - Call `draw` on each object using `SpaceObject` pointers
    - Proper `draw` function is called based on object's type
  - A new class derived from `SpaceObject` can be added without affecting the screen manager

Software Engineering Observation 13.1

With `virtual` functions and polymorphism, you can deal in generalities and let the execution-time environment concern itself with the specifics. You can direct a variety of objects to behave in manners appropriate to those objects without even knowing their types (as long as those objects belong to the same inheritance hierarchy and are being accessed off a common base-class pointer).

Software Engineering Observation 13.2

Polymorphism promotes extensibility: Software written to invoke polymorphic behavior is written independently of the types of the objects to which messages are sent. Thus, new types of objects that can respond to existing messages can be incorporated into such a system without modifying the base system. Only client code that instantiates new objects must be modified to accommodate new types.

13.3 Relationships Among Objects in an Inheritance Hierarchy

- Demonstration
  - Invoking base-class functions from derived-class objects
  - Aiming derived-class pointers at base-class objects
  - Derived-class member-function calls via base-class pointers
  - Demonstrating polymorphism using virtual functions
    - Base-class pointers aimed at derived-class objects
- Key concept
  - An object of a derived class can be treated as an object of its base class

13.3.1 Invoking Base-Class Functions from Derived-Class Objects

- Aim base-class pointer at base-class object
  - Invoke base-class functionality
- Aim derived-class pointer at derived-class object
  - Invoke derived-class functionality
- Aim base-class pointer at derived-class object
  - Because derived-class object *is an* object of base class
  - Invoke 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
  - `virtual` functions
    - Make it possible to invoke the object type’s functionality, rather than invoke the handle type’s functionality
    - Crucial to implementing polymorphic behavior

Outline

CommissionEmployee.h

(1 of 2)

```
// Fig. 13.1: CommissionEmployee.h
// CommissionEmployee class definition represents a commission employee.
#ifndef COMMISSION_H
#define COMMISSION_H

#include <string> // C++ standard string class
using std::string;

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

   void setFirstName( const string & ); // set first name
   string getFirstName() const; // return first name

   void setLastName( const string & ); // set last name
   string getLastName() const; // return last name

   void setSocialSecurityNumber( const string & ); // set SSN
   string getSocialSecurityNumber() const; // return SSN

   void setGrossSales( double ); // set gross sales amount
   double getGrossSales() const; // return gross sales amount
```

Outline

```
   void setCommissionRate( double ); // set commission rate
   double getCommissionRate() const; // return commission rate

   double earnings() const; // calculate earnings
   void print() const; // print CommissionEmployee object
private:
   string firstName;
   string lastName;
   string socialSecurityNumber;
   double grossSales; // gross weekly sales
   double commissionRate; // commission percentage
}; // end class CommissionEmployee

#endif
```

Function **`earnings`** will be redefined in derived classes to calculate the employee's earnings

Employee.h

(2 of 2)

Function **`print`** will be redefined in derived class to print the employee's information

```
// Fig. 13.2: CommissionEmployee.cpp
// Class CommissionEmployee member-function definitions.
#include <iostream>
using std::cout;

#include "CommissionEmployee.h" // CommissionEmployee class definition

// constructor
CommissionEmployee::CommissionEmployee(
   const string &first, const string &last, const string &ssn,
   double sales, double rate )
   : firstName( first ), lastName( last ), socialSecurityNumber( ssn )
{
   setGrossSales( sales ); // validate and store gross sales
   setCommissionRate( rate ); // validate and store commission rate
} // end CommissionEmployee constructor

// set first name
void CommissionEmployee::setFirstName( const string &first )
{
   firstName = first; // should validate
} // end function setFirstName

// return first name
string CommissionEmployee::getFirstName() const
{
   return firstName;
} // end function getFirstName
```

`Commission Employee.cpp`

(1 of 4)

```
// set last name
void CommissionEmployee::setLastName( const string &last )
{
   lastName = last;  // should validate
} // end function setLastName

// return last name
string CommissionEmployee::getLastName() const
{
   return lastName;
} // end function getLastName

// set social security number
void CommissionEmployee::setSocialSecurityNumber( const string &ssn )
{
   socialSecurityNumber = ssn; // should validate
} // end function setSocialSecurityNumber

// return social security number
string CommissionEmployee::getSocialSecurityNumber() const
{
   return socialSecurityNumber;
} // end function getSocialSecurityNumber

// set gross sales amount
void CommissionEmployee::setGrossSales( double sales )
{
   grossSales = ( sales < 0.0 ) ? 0.0 : sales;
} // end function setGrossSales
```


`CommissionEmployee.cpp`

```

// return gross sales amount
double CommissionEmployee::getGrossSales() const
{
   return grossSales;
} // end function getGrossSales

// set commission rate
void CommissionEmployee::setCommissionRate( double rate )
{
   commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
} // end function setCommissionRate

// return commission rate
double CommissionEmployee::getCommissionRate() const
{
   return commissionRate;
} // end function getCommissionRate

// calculate earnings
double CommissionEmployee::earnings() const
{
   return getCommissionRate() * getGrossSales();
} // end function earnings
```

Calculate earnings based on commission rate and gross sales

CommissionEmployee.cpp

(3 of 4)

```cpp

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

Display name, social security number, gross sales and commission rate

Outline

```
// Fig. 13.3: BasePlusCommissionEmployee.h
// BasePlusCommissionEmployee class derived from class
// CommissionEmployee.
#ifndef BASEPLUS_H
#define BASEPLUS_H

#include <string> // C++ standard string class
using std::string;

#include "CommissionEmployee.h" // CommissionEmployee class declaration

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

   void setBaseSalary( double ); // set base salary
   double getBaseSalary() const; // return base salary

   double earnings() const; // calculate earnings
   void print() const; // print BasePlusCommissionEmployee object
private:
   double baseSalary; // base salary
}; // end class BasePlusCommissionEmployee

#endif
```

Redefine functions **`earnings`** and **`print`**

`BasePlus Commission Employee.h`

(1 of 1)

```
// Fig. 13.4: BasePlusCommissionEmployee.cpp
// Class BasePlusCommissionEmployee member-function definitions.
#include <iostream>
using std::cout;

// BasePlusCommissionEmployee class definition
#include "BasePlusCommissionEmployee.h"

// constructor
BasePlusCommissionEmployee::BasePlusCommissionEmployee(
   const string &first, const string &last, const string &ssn,
   double sales, double rate, double salary )
   // explicitly call base-class constructor
   : CommissionEmployee( first, last, ssn, sales, rate )
{
   setBaseSalary( salary ); // validate and store base salary
} // end BasePlusCommissionEmployee constructor

// set base salary
void BasePlusCommissionEmployee::setBaseSalary( double salary )
{
   baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
} // end function setBaseSalary

// return base salary
double BasePlusCommissionEmployee::getBaseSalary() const
{
   return baseSalary;
} // end function getBaseSalary
```

Outline

BasePlus Commission Employee.cpp

Outline

BasePlus Commission ….cpp

(2 of 2)

```
// calculate earnings
double BasePlusCommissionEmployee::earnings() const
{
   return getBaseSalary() + CommissionEmployee::earnings();
} // end function earnings

// print BasePlusCommissionEmployee object
void BasePlusCommissionEmployee::print() const
{
   cout << "base-salaried ";

   // invoke CommissionEmployee's print function
   CommissionEmployee::print();

   cout << "\nbase salary: " << getBaseSalary();
} // end function print
```

Redefined `earnings` function incorporates base salary

Redefined `print` function displays additional **`BasePlusCommissionEmployee`** details

Outline

fig13_05.cpp

(1 of 5)

```
// Fig. 13.5: fig13_05.cpp
// Aiming base-class and derived-class pointers at base-class
// and derived-class objects, respectively.
#include <iostream>
using std::cout;
using std::endl;
using std::fixed;

#include <iomanip>
using std::setprecision;

// include class definitions
#include "CommissionEmployee.h"
#include "BasePlusCommissionEmployee.h"

int main()
{
   // create base-class object
   CommissionEmployee commissionEmployee(
      "Sue", "Jones", "222-22-2222", 10000, .06 );

   // create base-class pointer
   CommissionEmployee *commissionEmployeePtr = 0;
```

Outline

fig13_05.cpp


```
   // create derived-class object
   BasePlusCommissionEmployee basePlusCommissionEmployee(
      "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );

   // create derived-class pointer
   BasePlusCommissionEmployee *basePlusCommissionEmployeePtr = 0;

   // set floating-point output formatting
   cout << fixed << setprecision( 2 );

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

   // aim base-class pointer at base-class object and print
   commissionEmployeePtr = &commissionEmployee; // perfectly natural
   cout << "\n\n\nCalling print with base-class pointer to "
      << "\nbase-class object invokes base-class print function:\n\n";
   commissionEmployeePtr->print(); // invokes base-clas
```

Aiming base-class pointer at base-class object and invoking base-class functionality

Outline

```
    // aim derived-class pointer at derived-class object and print
    basePlusCommissionEmployeePtr = &basePlusCommissionEmployee; // natural
    cout << "\n\n\nCalling print with derived-class pointer to "
       << "\nderived-class object invokes derived-class "
       << "print function:\n\n";
    basePlusCommissionEmployeePtr->print(); // invokes derived-class print

    // aim base-class pointer at derived-class object and print
    commissionEmployeePtr = &basePlusCommissionEmployee;
    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(); // invokes base-class print
    cout << endl;
    return 0;
 } // end main
```

fig13_05.cpp

Aiming derived-class pointer at derived-class object and invoking derived-class functionality

Aiming base-class pointer at derived-class object and invoking base-class functionality

Outline

```
Print base-class and derived-class objects:

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00


Calling print with base-class pointer to
base-class object invokes base-class print function:

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06
```

(Continued at top of next slide…)

`fig13_05.cpp`

(…Continued from bottom of previous slide)

```
Calling print with derived-class pointer to
derived-class object invokes derived-class print function:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00


Calling print with base-class pointer to derived-class object
invokes base-class print function on that derived-class object:

commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
```

13.3.2 Aiming Derived-Class Pointers at Base-Class Objects

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

Outline

```
// Fig. 13.6: fig13_06.cpp
// Aiming a derived-class pointer at a base-class object.
#include "CommissionEmployee.h"
#include "BasePlusCommissionEmployee.h"

int main()
{
   CommissionEmployee commissionEmployee(
      "Sue", "Jones", "222-22-2222", 10000, .06 );
   BasePlusCommissionEmployee *basePlusCommissionEmployeePtr = 0;

   // aim derived-class pointer at base-class object
   // Error: a CommissionEmployee is not a BasePlusCommissionEmployee
   basePlusCommissionEmployeePtr = &commissionEmployee;
   return 0;
} // end main
```

fig13_06.cpp

(1 of 2)

Cannot assign base-class object to derived-class pointer because *is-a* relationship does not apply

Borland C++ command-line compiler error messages:

```
Error E2034 Fig13_06\fig13_06.cpp 14: Cannot convert 'CommissionEmployee *'
   to 'BasePlusCommissionEmployee *' in function main()
```

GNU C++ compiler error messages:

```
fig13_06.cpp:14: error: invalid conversion from `CommissionEmployee*' to
   `BasePlusCommissionEmployee*'
```

Microsoft Visual C++.NET compiler error messages:

```
C:\cpphtp5_examples\ch13\Fig13_06\fig13_06.cpp(14) : error C2440:
   '=' : cannot convert from 'CommissionEmployee *__w64 ' to
   'BasePlusCommissionEmployee *'
         Cast from base to derived requires dynamic_cast or static_cast
```

13.3.3 Derived-Class Member-Function Calls via Base-Class Pointers

- Aiming base-class pointer at derived-class object
  - 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) will result in error
    - Derived-class members cannot be accessed from base-class pointers
    - However, they can be accomplished using downcasting (Section 13.8)

```cpp
// Fig. 13.7: fig13_07.cpp
// Attempting to invoke derived-class-only member functions
// through a base-class pointer.
#include "CommissionEmployee.h"
#include "BasePlusCommissionEmployee.h"

int main()
{
   CommissionEmployee *commissionEmployeePtr = 0; // base class
   BasePlusCommissionEmployee basePlusCommissionEmployee(
      "Bob", "Lewis", "333-33-3333", 5000, .04, 300 ); // derived class

   // aim base-class pointer at derived-class object
   commissionEmployeePtr = &basePlusCommissionEmployee;

   // 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 );
   return 0;
} // end main
```

Cannot invoke derived-class-only members from base-class pointer

Outline

Borland C++ command-line compiler error messages:

```
Error E2316 Fig13_07\fig13_07.cpp 26: 'getBaseSalary' is not a member of
   'CommissionEmployee' in function main()
Error E2316 Fig13_07\fig13_07.cpp 27: 'setBaseSalary' is not a member of
   'CommissionEmployee' in function main()
```

Microsoft Visual C++.NET compiler error messages:

```
C:\cpphtp5_examples\ch13\Fig13_07\fig13_07.cpp(26) : error C2039:
   'getBaseSalary' : is not a member of 'CommissionEmployee'
      C:\cpphtp5_examples\ch13\Fig13_07\CommissionEmployee.h(10) :
         see declaration of 'CommissionEmployee'
C:\cpphtp5_examples\ch13\Fig13_07\fig13_07.cpp(27) : error C2039:
   'setBaseSalary' : is not a member of 'CommissionEmployee'
      C:\cpphtp5_examples\ch13\Fig13_07\CommissionEmployee.h(10) :
         see declaration of 'CommissionEmployee'
```

GNU C++ compiler error messages:

```
fig13_07.cpp:26: error: `getBaseSalary' undeclared (first use this function)
fig13_07.cpp:26: error: (Each undeclared identifier is reported only once for
   each function it appears in.)
fig13_07.cpp:27: error: `setBaseSalary' undeclared (first use this function)
```

fig13_07.cpp

(2 of 2)

Software Engineering Observation 13.3

If the address of a derived-class object has been assigned to a pointer of one of its direct or indirect base classes, it is acceptable to cast that base-class pointer back to a pointer of the derived-class type. In fact, this must be done to send that derived-class object messages that do not appear in the base class.

13.3.4 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

13.3.4 Virtual Functions (Cont.)

- `virtual` functions
  - Declared by preceding the function's prototype with the keyword `virtual` in base class
  - Derived classes override function as appropriate
  - Once declared `virtual`, a function remains `virtual` all the way down the hierarchy
  - 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

Software Engineering Observation 13.4

Once a function is declared `virtual`, it remains `virtual` all the way down the inheritance hierarchy from that point, even if that function is not explicitly declared `virtual` when a class overrides it.

Good Programming Practice 13.1

Even though certain functions are implicitly `virtual` because of a declaration made higher in the class hierarchy, explicitly declare these functions `virtual` at every level of the hierarchy to promote program clarity.

Error-Prevention Tip 13.1

When a programmer browses a class hierarchy to locate a class to reuse, it is possible that a function in that class will exhibit `virtual` function behavior even though it is not explicitly declared `virtual`. This happens when the class inherits a `virtual` function from its base class, and it can lead to subtle logic errors. Such errors can be avoided by explicitly declaring all `virtual` functions `virtual` throughout the inheritance hierarchy.

Software Engineering Observation 13.5

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.

Outline

```
// Fig. 13.8: CommissionEmployee.h
// CommissionEmployee class definition represents a commission employee.
#ifndef COMMISSION_H
#define COMMISSION_H

#include <string> // C++ standard string class
using std::string;

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

   void setFirstName( const string & ); // set first name
   string getFirstName() const; // return first name

   void setLastName( const string & ); // set last name
   string getLastName() const; // return last name

   void setSocialSecurityNumber( const string & ); // set SSN
   string getSocialSecurityNumber() const; // return SSN

   void setGrossSales( double ); // set gross sales amount
   double getGrossSales() const; // return gross sales amount
```

Commission Employee.h

(1 of 2)

```
   void setCommissionRate( double ); // set commission rate
   double getCommissionRate() const; // return commission rate

   virtual double earnings() const; // calculate earnings
   virtual void print() const; // print CommissionEmployee object
private:
   string firstName;
   string lastName;
   string socialSecurityNumber;
   double grossSales; // gross weekly sales
   double commissionRate; // commission percentage
}; // end class CommissionEmployee

#endif
```

Declaring **earnings** and **print** as **virtual** allows them to be overridden, not redefined

```
// Fig. 13.9: BasePlusCommissionEmployee.h
// BasePlusCommissionEmployee class derived from class
// CommissionEmployee.
#ifndef BASEPLUS_H
#define BASEPLUS_H

#include <string> // C++ standard string class
using std::string;

#include "CommissionEmployee.h" // CommissionEmployee class declaration

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

   void setBaseSalary( double ); // set base salary
   double getBaseSalary() const; // return base salary

   virtual double earnings() const; // calculate earnings
   virtual void print() const; // print BasePlusCommissionEmployee object
private:
   double baseSalary; // base salary
}; // end class BasePlusCommissionEmployee

#endif
```

Functions **`earnings`** and **`print`** are already **`virtual`** – good practice to declare **`virtual`** even when overriding function

`BasePlus Commission Employee.h`

```
// Fig. 13.10: fig13_10.cpp
// Introducing polymorphism, virtual functions and dynamic binding.
#include <iostream>
using std::cout;
using std::endl;
using std::fixed;

#include <iomanip>
using std::setprecision;

// include class definitions
#include "CommissionEmployee.h"
#include "BasePlusCommissionEmployee.h"

int main()
{
   // create base-class object
   CommissionEmployee commissionEmployee(
      "Sue", "Jones", "222-22-2222", 10000, .06 );

   // create base-class pointer
   CommissionEmployee *commissionEmployeePtr = 0;

   // create derived-class object
   BasePlusCommissionEmployee basePlusCommissionEmployee(
      "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );

   // create derived-class pointer
   BasePlusCommissionEmployee *basePlusCommissionEmployeePtr = 0;
```

Outline

fig13_10.cpp

(1 of 5)

```

   // set floating-point output formatting
   cout << fixed << setprecision( 2 );

   // output objects using static binding
   cout << "Invoking print function on base-class and derived-class "
      << "\nobjects with static binding\n\n";
   commissionEmployee.print(); // static binding
   cout << "\n\n";
   basePlusCommissionEmployee.print(); // static binding

   // output objects using dynamic binding
   cout << "\n\n\nInvoking print function on base-class and "
      << "derived-class \nobjects with dynamic binding";

   // aim base-class pointer at base-class object and print
   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
```

Aiming base-class pointer at base-class object and invoking base-class functionality

```
   // aim derived-class pointer at derived-class object and 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

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

   // polymorphism; invokes BasePlusCommissionEmployee's print;
   // base-class pointer to derived-class object
   commissionEmployeePtr->print();
   cout << endl;
   return 0;
} // end main
```

Aiming derived-class pointer at derived-class object and invoking derived-class functionality

Aiming base-class pointer at derived-class object and invoking derived-class functionality via polymorphism and **`virtual`** functions

```
Invoking print function on base-class and derived-class
objects with static binding

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00


Invoking print function on base-class and derived-class
objects with dynamic binding

Calling virtual function print with base-class pointer
to base-class object invokes base-class print function:

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06

Calling virtual function print with derived-class pointer
to derived-class object invokes derived-class print function:
```

(Coninued at the top of next slide …)

fig13_10.cpp

(…Continued from the bottom of previous slide)

```
base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00

Calling virtual function print with base-class pointer
to derived-class object invokes derived-class print function:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00
```

13.3.5 Summary of the Allowed Assignments Between Base-Class and Derived-Class Objects and Pointers

- Four ways to aim base-class and derived-class pointers at base-class and derived-class objects
  - Aiming a base-class pointer at a base-class object
    - Is straightforward
  - Aiming a derived-class pointer at a derived-class object
    - Is straightforward
  - Aiming a base-class pointer at a derived-class object
    - Is safe, but can be used to invoke only member functions that base-class declares (unless downcasting is used)
    - Can achieve polymorphism with `virtual` functions
  - Aiming a derived-class pointer at a base-class object
    - Generates a compilation error

Common Programming Error 13.1

After aiming a base-class pointer at a derived-class object, attempting to reference derived-class-only members with the base-class pointer is a compilation error.

Common Programming Error 13.2

Treating a base-class object as a derived-class object can cause errors.

13.4 Type Fields and `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

Software Engineering Observation 13.6

Polymorphic programming can eliminate the need for unnecessary `switch` logic. By using the C++ polymorphism mechanism to perform the equivalent logic, programmers can avoid the kinds of errors typically associated with `switch` logic.

Software Engineering Observation 13.7

An interesting consequence of using polymorphism is that programs take on a simplified appearance. They contain less branching logic and more simple, sequential code. This simplification facilitates testing, debugging and program maintenance.

13.5 Abstract Classes and Pure `virtual` Functions

- Abstract classes
  - Classes from which the programmer never intends to instantiate any objects
    - 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
    - Classes used to instantiate objects are called concrete classes
      - Must provide implementation for every member function they define

13.5 Abstract Classes and Pure `virtual` Functions (Cont.)

- Pure `virtual` function
  - A class is made abstract by declaring one or more of its `virtual` functions to be "pure"
    - Placing "`= 0`" in its declaration
      - Example
        - `virtual void draw() const = 0;`
      - "`= 0`" is known as a pure specifier
  - Do not provide implementations
    - Every concrete derived class must override all base-class pure `virtual` functions with concrete implementations
      - 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

Software Engineering Observation 13.8

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.

Common Programming Error 13.3

Attempting to instantiate an object of an abstract class causes a compilation error.

Common Programming Error 13.4

Failure to override a pure `virtual` function in a derived class, then attempting to instantiate objects of that class, is a compilation error.

Software Engineering Observation 13.9

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), which are subject to the normal rules of inheritance by derived classes.

13.5 Abstract Classes and Pure `virtual` Functions (Cont.)

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

13.6 Case Study: Payroll System Using Polymorphism

- Enhanced `CommissionEmployee`-`BasePlusCommissionEmployee` hierarchy using an abstract base class
  - Abstract class `Employee` represents the general concept of an employee
    - Declares the "interface" to the hierarchy
    - Each employee has a first name, last name and social security number
  - Earnings calculated differently and objects printed differently for each derived classe

Software Engineering Observation 13.10

A derived class can inherit interface or implementation from a base class. 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. 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).

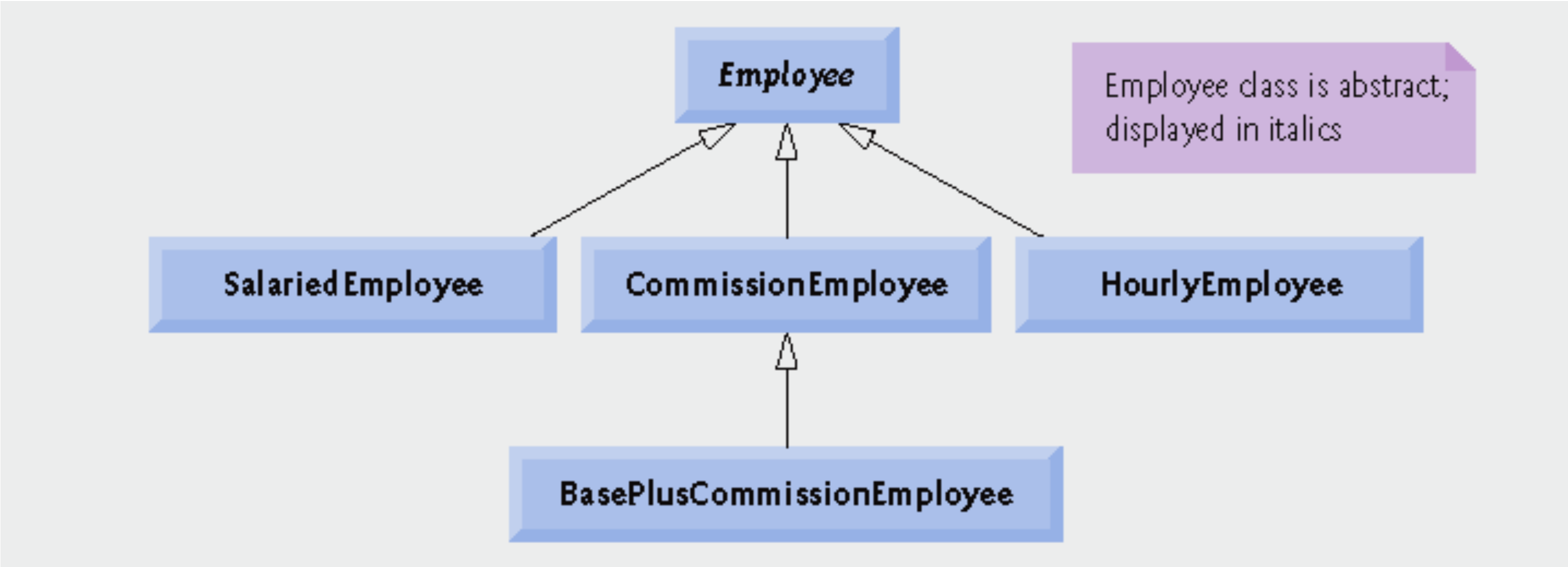

Fig.13.11 | Employee hierarchy UML class diagram.

13.6.1 Creating Abstract Base Class `Employee`

- Class `Employee`
  - Provides various *get* and *set* functions
  - Provides functions `earnings` and `print`
    - Function `earnings` depends on type of employee, so declared pure `virtual`
      - Not enough information in class `Employee` for a default implementation
    - Function `print` is virtual, but not pure `virtual`
      - Default implementation provided in `Employee`
  - Example maintains a `vector` of `Employee` pointers
    - Polymorphically invokes proper `earnings` and `print` functions

	earnings	print
Employee	= 0	*firstName lastName* social security number: *SSN*
Salaried-Employee	weeklySalary	salaried employee: *firstName lastName* social security number: *SSN* weekly salary: *weeklysalary*
Hourly-Employee	*If hours <= 40* wage * hours *If hours > 40* (40 * wage) + ((hours - 40) * wage * 1.5)	hourly employee: *firstName lastName* social security number: *SSN* hourly wage: *wage*; hours worked: *hours*
Commission-Employee	commissionRate * grossSales	commission employee: *firstName lastName* social security number: *SSN* gross sales: *grossSales*; commission rate: *commissionRate*
BasePlus-Commission-Employee	baseSalary + (commissionRate * grossSales)	base salaried commission employee: *firstName lastName* social security number: *SSN* gross sales: *grossSales*; commission rate: *commissionRate*; base salary: *baseSalary*

Fig.13.12 | **Polymorphic interface for the `Employee` hierarchy classes.**

Outline

```
// Fig. 13.13: Employee.h
// Employee abstract base class.
#ifndef EMPLOYEE_H
#define EMPLOYEE_H

#include <string> // C++ standard string class
using std::string;

class Employee
{
public:
   Employee( const string &, const string &, const string & );

   void setFirstName( const string & ); // set first name
   string getFirstName() const; // return first name

   void setLastName( const string & ); // set last name
   string getLastName() const; // return last name

   void setSocialSecurityNumber( const string & ); // set SSN
   string getSocialSecurityNumber() const; // return SSN
```

Employee.h

(1 of 2)

Outline

```
   // pure virtual function makes Employee abstract base class
   virtual double earnings() const = 0; // pure virtual
   virtual void print() const; // virtual
private:
   string firstName;
   string lastName;
   string socialSecurityNumber;
}; // end class Employee

#endif // EMPLOYEE_H
```

Function **earnings** is pure **virtual**, not enough data to provide a default, concrete implementation

(2 of 2)

Function **print** is **virtual**, default implementation provided but derived-classes may override

Outline

```
// Fig. 13.14: Employee.cpp
// Abstract-base-class Employee member-function definitions.
// Note: No definitions are given for pure virtual functions.
#include <iostream>
using std::cout;

#include "Employee.h" // Employee class definition

// constructor
Employee::Employee( const string &first, const string &last,
   const string &ssn )
   : firstName( first ), lastName( last ), socialSecurityNumber( ssn )
{
   // empty body
} // end Employee constructor

// set first name
void Employee::setFirstName( const string &first )
{
   firstName = first;
} // end function setFirstName

// return first name
string Employee::getFirstName() const
{
   return firstName;
} // end function getFirstName

```

Employee.cpp

(1 of 2)

Outline

```cpp
// set last name
void Employee::setLastName( const string &last )
{
   lastName = last;
} // end function setLastName

// return last name
string Employee::getLastName() const
{
   return lastName;
} // end function getLastName

// set social security number
void Employee::setSocialSecurityNumber( const string &ssn )
{
   socialSecurityNumber = ssn; // should validate
} // end function setSocialSecurityNumber

// return social security number
string Employee::getSocialSecurityNumber() const
{
   return socialSecurityNumber;
} // end function getSocialSecurityNumber

// print Employee's information (virtual, but not pure virtual)
void Employee::print() const
{
   cout << getFirstName() << ' ' << getLastName()
      << "\nsocial security number: " << getSocialSecurityNumber();
} // end function print
```

Employee.cpp

13.6.2 Creating Concrete Derived Class `SalariedEmployee`

- `SalariedEmployee` inherits from `Employee`
  - Includes a weekly salary
    - Overridden `earnings` function incorporates weekly salary
    - Overridden `print` function incorporates weekly salary
  - Is a concrete class (implements all pure `virtual` functions in abstract base class)

Outline

SalariedEmployee.h

```
// Fig. 13.15: SalariedEmployee.h
// SalariedEmployee class derived from Employee.
#ifndef SALARIED_H
#define SALARIED_H

#include "Employee.h" // Employee class definition

class SalariedEmployee : public Employee
{
public:
   SalariedEmployee( const string &, const string &,
      const string &, double = 0.0 );

   void setWeeklySalary( double ); // set weekly salary
   double getWeeklySalary() const; // return weekly salary

   // keyword virtual signals intent to override
   virtual double earnings() const; // calculate earnings
   virtual void print() const; // print SalariedEmployee object
private:
   double weeklySalary; // salary per week
}; // end class SalariedEmployee

#endif // SALARIED_H
```

SalariedEmployee inherits from **Employee**, must override **earnings** to be concrete

Functions will be overridden (or defined for the first time)

```
// Fig. 13.16: SalariedEmployee.cpp
// SalariedEmployee class member-function definitions.
#include <iostream>
using std::cout;

#include "SalariedEmployee.h" // SalariedEmployee class definition

// constructor
SalariedEmployee::SalariedEmployee( const string &first,
   const string &last, const string &ssn, double salary )
   : Employee( first, last, ssn )
{
   setWeeklySalary( salary );
} // end SalariedEmployee constructor

// set salary
void SalariedEmployee::setWeeklySalary( double salary )
{
   weeklySalary = ( salary < 0.0 ) ? 0.0 : salary;
} // end function setWeeklySalary

// return salary
double SalariedEmployee::getWeeklySalary() const
{
   return weeklySalary;
} // end function getWeeklySalary
```

Maintain new data member **weeklySalary**

Outline

SalariedEmployee.cpp

(1 of 2)

Outline

SalariedEmployee.cpp


```

// calculate earnings;
// override pure virtual function earnings in Employee
double SalariedEmployee::earnings() const
{
   return getWeeklySalary();
} // end function earnings

// print SalariedEmployee's information
void SalariedEmployee::print() const
{
   cout << "salaried employee: ";
   Employee::print(); // reuse abstract base-class print function
   cout << "\nweekly salary: " << getWeeklySalary();
} // end function print
```

Overridden `earnings` and `print` functions incorporate weekly salary

13.6.3 Creating Concrete Derived Class `HourlyEmployee`

- `HourlyEmployee` inherits from `Employee`
  - Includes a wage and hours worked
    - Overridden `earnings` function incorporates the employee’s wages multiplied by hours (taking time-and-a-half pay into account)
    - Overridden `print` function incorporates wage and hours worked
  - Is a concrete class (implements all pure `virtual` functions in abstract base class)

Outline

Hourly
Employee.h

```
// Fig. 13.17: HourlyEmployee.h
// HourlyEmployee class definition.
#ifndef HOURLY_H
#define HOURLY_H

#include "Employee.h" // Employee class definition

class HourlyEmployee : public Employee
{
public:
   HourlyEmployee( const string &, const string &,
      const string &, double = 0.0, double = 0.0 );

   void setWage( double ); // set hourly wage
   double getWage() const; // return hourly wage

   void setHours( double ); // set hours worked
   double getHours() const; // return hours worked

   // keyword virtual signals intent to override
   virtual double earnings() const; // calculate earnings
   virtual void print() const; // print HourlyEmployee object
private:
   double wage; // wage per hour
   double hours; // hours worked for week
}; // end class HourlyEmployee

#endif // HOURLY_H
```

`HourlyEmployee` inherits from **`Employee`**, must override **`earnings`** to be concrete

Functions will be overridden (or defined for first time)

Outline

```
// Fig. 13.18: HourlyEmployee.cpp
// HourlyEmployee class member-function definitions.
#include <iostream>
using std::cout;

#include "HourlyEmployee.h" // HourlyEmployee class definition

// constructor
HourlyEmployee::HourlyEmployee( const string &first, const string &last,
   const string &ssn, double hourlyWage, double hoursWorked )
   : Employee( first, last, ssn )
{
   setWage( hourlyWage ); // validate hourly wage
   setHours( hoursWorked ); // validate hours worked
} // end HourlyEmployee constructor

// set wage
void HourlyEmployee::setWage( double hourlyWage )
{
   wage = ( hourlyWage < 0.0 ? 0.0 : hourlyWage );
} // end function setWage

// return wage
double HourlyEmployee::getWage() const
{
   return wage;
} // end function getWage
```

Maintain new data member, **`hourlyWage`**

`Hourly Employee.cpp`

(1 of 2)

```

// set hours worked
void HourlyEmployee::setHours( double hoursWorked )
{
   hours = ( ( ( hoursWorked >= 0.0 ) && ( hoursWorked <= 168.0 ) ) ?
      hoursWorked : 0.0 );
} // end function setHours

// return hours worked
double HourlyEmployee::getHours() const
{
   return hours;
} // end function getHours

// calculate earnings;
// override pure virtual function earnings in Employee
double HourlyEmployee::earnings() const
{
   if ( getHours() <= 40 ) // no overtime
      return getWage() * getHours();
   else
      return 40 * getWage() + ( ( getHours() - 40 ) * getWage() * 1.5 );
} // end function earnings

// print HourlyEmployee's information
void HourlyEmployee::print() const
{
   cout << "hourly employee: ";
   Employee::print(); // code reuse
   cout << "\nhourly wage: " << getWage() <<
      "; hours worked: " << getHours();
} // end function print
```

Maintain new data member, **hoursWorked**

Overridden **earnings** and **print** functions incorporate wage and hours

13.6.4 Creating Concrete Derived Class `CommissionEmployee`

- `CommissionEmployee` inherits from `Employee`
  - Includes gross sales and commission rate
    - Overridden `earnings` function incorporates gross sales and commission rate
    - Overridden `print` function incorporates gross sales and commission rate
  - Concrete class (implements all pure `virtual` functions in abstract base class)

Outline

```
// Fig. 13.19: CommissionEmployee.h
// CommissionEmployee class derived from Employee.
#ifndef COMMISSION_H
#define COMMISSION_H

#include "Employee.h" // Employee class definition

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

   void setCommissionRate( double ); // set commission rate
   double getCommissionRate() const; // return commission rate

   void setGrossSales( double ); // set gross sales amount
   double getGrossSales() const; // return gross sales amount

   // keyword virtual signals intent to override
   virtual double earnings() const; // calculate earnings
   virtual void print() const; // print CommissionEmployee object
private:
   double grossSales; // gross weekly sales
   double commissionRate; // commission percentage
}; // end class CommissionEmployee

#endif // COMMISSION_H
```

Commission Employee.h

CommissionEmployee inherits from **Employee**, must override **earnings** to be concrete

Functions will be overridden (or defined for first time)

```
// Fig. 13.20: CommissionEmployee.cpp
// CommissionEmployee class member-function definitions.
#include <iostream>
using std::cout;

#include "CommissionEmployee.h" // CommissionEmployee class definition

// constructor
CommissionEmployee::CommissionEmployee( const string &first,
   const string &last, const string &ssn, double sales, double rate )
   : Employee( first, last, ssn )
{
   setGrossSales( sales );
   setCommissionRate( rate );
} // end CommissionEmployee constructor

// set commission rate
void CommissionEmployee::setCommissionRate( double rate )
{
   commissionRate = ( ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0 );
} // end function setCommissionRate

// return commission rate
double CommissionEmployee::getCommissionRate() const
{
    return commissionRate;
} // end function getCommissionRate
```

Outline

CommissionEmployee.cpp

(1 of 2)

Maintain new data member, **commissionRate**

Outline

sion
ee.cpp

(2 of 2)

```

// set gross sales amount
void CommissionEmployee::setGrossSales( double sales )
{
   grossSales = ( ( sales < 0.0 ) ? 0.0 : sales );
} // end function setGrossSales

// return gross sales amount
double CommissionEmployee::getGrossSales() const
{
    return grossSales;
} // end function getGrossSales

// calculate earnings;
// override pure virtual function earnings in Employee
double CommissionEmployee::earnings() const
{
   return getCommissionRate() * getGrossSales();
} // end function earnings

// print CommissionEmployee's information
void CommissionEmployee::print() const
{
   cout << "commission employee: ";
   Employee::print(); // code reuse
   cout << "\ngross sales: " << getGrossSales()
      << "; commission rate: " << getCommissionRate();
} // end function print
```

Maintain new data member, **grossSales**

Overridden **earnings** and **print** functions incorporate commission rate and gross sales

13.6.5 Creating Indirect Concrete Derived Class `BasePlusCommissionEmployee`

- `BasePlusCommissionEmployee` inherits from `CommissionEmployee`
  - Includes base salary
    - Overridden `earnings` function that incorporates base salary
    - Overridden `print` function that incorporates base salary
  - Concrete class, because derived class is concrete
    - Not necessary to override `earnings` to make it concrete, can inherit implementation from `CommissionEmployee`
      - Although we do override `earnings` to incorporate base salary

Outline

BasePlusCommissionEmployee.h

```cpp
// Fig. 13.21: BasePlusCommissionEmployee.h
// BasePlusCommissionEmployee class derived from Employee.
#ifndef BASEPLUS_H
#define BASEPLUS_H

#include "CommissionEmployee.h" // CommissionEmployee class definition

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

   void setBaseSalary( double ); // set base salary
   double getBaseSalary() const; // return base salary

   // keyword virtual signals intent to override
   virtual double earnings() const; // calculate earnings
   virtual void print() const; // print BasePlusCommissionEmployee object
private:
   double baseSalary; // base salary per week
}; // end class BasePlusCommissionEmployee

#endif // BASEPLUS_H
```

BasePlusCommissionEmployee inherits from **CommissionEmployee**, already concrete

Functions will be overridden

Outline

```
// Fig. 13.22: BasePlusCommissionEmployee.cpp
// BasePlusCommissionEmployee member-function definitions.
#include <iostream>
using std::cout;

// BasePlusCommissionEmployee class definition
#include "BasePlusCommissionEmployee.h"

// constructor
BasePlusCommissionEmployee::BasePlusCommissionEmployee(
   const string &first, const string &last, const string &ssn,
   double sales, double rate, double salary )
   : CommissionEmployee( first, last, ssn, sales, rate )
{
   setBaseSalary( salary ); // validate and store base salary
} // end BasePlusCommissionEmployee constructor

// set base salary
void BasePlusCommissionEmployee::setBaseSalary( double salary )
{
   baseSalary = ( ( salary < 0.0 ) ? 0.0 : salary );
} // end function setBaseSalary

// return base salary
double BasePlusCommissionEmployee::getBaseSalary() const
{
    return baseSalary;
} // end function getBaseSalary
```

Maintain new data member, **`baseSalary`**

`BasePlus Commission Employee.cpp`

(1 of 2)

```
// calculate earnings;
// override pure virtual function earnings in Employee
double BasePlusCommissionEmployee::earnings() const
{
    return getBaseSalary() + CommissionEmployee::earnings();
} // end function earnings

// print BasePlusCommissionEmployee's information
void BasePlusCommissionEmployee::print() const
{
   cout << "base-salaried ";
   CommissionEmployee::print(); // code reuse
   cout << "; base salary: " << getBaseSalary();
} // end function print
```

Outline

BasePlus Commission Employee.cpp

Overridden **earnings** and **print** functions incorporate base salary

13.6.6 Demonstrating Polymorphic Processing

- Create objects of types `SalariedEmployee`, `HourlyEmployee`, `CommissionEmployee` and `BasePlusCommissionEmployee`
  - Demonstrate manipulating objects with static binding
    - Using name handles rather than pointers or references
    - Compiler can identify each object's type to determine which `print` and `earnings` functions to call
  - Demonstrate manipulating objects polymorphically
    - Uses a `vector` of `Employee` pointers
    - Invoke `virtual` functions using pointers and references

Outline

fig13_23.cpp

(1 of 7)

```
// Fig. 13.23: fig13_23.cpp
// Processing Employee derived-class objects individually
// and polymorphically using dynamic binding.
#include <iostream>
using std::cout;
using std::endl;
using std::fixed;

#include <iomanip>
using std::setprecision;

#include <vector>
using std::vector;

// include definitions of classes in Employee hierarchy
#include "Employee.h"
#include "SalariedEmployee.h"
#include "HourlyEmployee.h"
#include "CommissionEmployee.h"
#include "BasePlusCommissionEmployee.h"

void virtualViaPointer( const Employee * const ); // prototype
void virtualViaReference( const Employee & ); // prototype
```

Outline

fig13_23.cpp

(2 of 7)

```

int main()
{
   // set floating-point output formatting
   cout << fixed << setprecision( 2 );

   // create derived-class objects
   SalariedEmployee salariedEmployee(
      "John", "Smith", "111-11-1111", 800 );
   HourlyEmployee hourlyEmployee(
      "Karen", "Price", "222-22-2222", 16.75, 40 );
   CommissionEmployee commissionEmployee(
      "Sue", "Jones", "333-33-3333", 10000, .06 );
   BasePlusCommissionEmployee basePlusCommissionEmployee(
      "Bob", "Lewis", "444-44-4444", 5000, .04, 300 );

   cout << "Employees processed individually using static binding:\n\n";

   // output each Employee's information and earnings using static binding
   salariedEmployee.print();
   cout << "\nearned $" << salariedEmployee.earnings() << "\n\n";
   hourlyEmployee.print();
   cout << "\nearned $" << hourlyEmployee.earnings() << "\n\n";
   commissionEmployee.print();
   cout << "\nearned $" << commissionEmployee.earnings() << "\n\n";
   basePlusCommissionEmployee.print();
   cout << "\nearned $" << basePlusCommissionEmployee.earnings()
      << "\n\n";
```

Using objects (rather than pointers or references) to demonstrate static binding

Outline

```
   // create vector of four base-class pointers
   vector < Employee * > employees( 4 );

   // initialize vector with Employees
   employees[ 0 ] = &salariedEmployee;
   employees[ 1 ] = &hourlyEmployee;
   employees[ 2 ] = &commissionEmployee;
   employees[ 3 ] = &basePlusCommissionEmployee;

   cout << "Employees processed polymorphically via dynamic binding:\n\n";

   // call virtualViaPointer to print each Employee's information
   // and earnings using dynamic binding
   cout << "Virtual function calls made off base-class pointers:\n\n";

   for ( size_t i = 0; i < employees.size(); i++ )
      virtualViaPointer( employees[ i ] );

   // call virtualViaReference to print each Employee's information
   // and earnings using dynamic binding
   cout << "Virtual function calls made off base-class references:\n\n";

   for ( size_t i = 0; i < employees.size(); i++ )
      virtualViaReference( *employees[ i ] ); // note dereferencing

   return 0;
} // end main
```

vector of **Employee** pointers, will be used to demonstrate dynamic binding

Demonstrate dynamic binding using first pointers, then references

Outline

fig13_23.cpp

(4 of 7)

```
// call Employee virtual functions print and earnings off a
// base-class pointer using dynamic binding
void virtualViaPointer( const Employee * const baseClassPtr )
{
   baseClassPtr->print();
   cout << "\nearned $" << baseClassPtr->earnings() << "\n\n";
} // end function virtualViaPointer

// call Employee virtual functions print and earnings off a
// base-class reference using dynamic binding
void virtualViaReference( const Employee &baseClassRef )
{
   baseClassRef.print();
   cout << "\nearned $" << baseClassRef.earnings() << "\n\n";
} // end function virtualViaReference
```

Using references and pointers cause **virtual** functions to be invoked polymorphically

```
Employees processed individually using static binding:

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: 800.00
earned $800.00

hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: 16.75; hours worked: 40.00
earned $670.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: 10000.00; commission rate: 0.06
earned $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: 5000.00; commission rate: 0.04; base salary: 300.00
earned $500.00
```

(Continued at top of next slide...)

(...continued from bottom of previous slide)

```
Employees processed polymorphically using dynamic binding:

Virtual function calls made off base-class pointers:

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: 800.00
earned $800.00

hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: 16.75; hours worked: 40.00
earned $670.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: 10000.00; commission rate: 0.06
earned $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: 5000.00; commission rate: 0.04; base salary: 300.00
earned $500.00
```

(Continued at the top of next slide...)

(...Continued from bottom of previous page)

```
Virtual function calls made off base-class references:

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: 800.00
earned $800.00

hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: 16.75; hours worked: 40.00
earned $670.00


commission employee: Sue Jones
social security number: 333-33-3333
gross sales: 10000.00; commission rate: 0.06
earned $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: 5000.00; commission rate: 0.04; base salary: 300.00
earned $500.00
```

13.7 (Optional) Polymorphism, Virtual Functions and Dynamic Binding "Under the Hood"

- How can C++ implement polymorphism, `virtual` functions and dynamic binding internally?
  - Three levels of pointers ("triple indirection")
  - Virtual function table (*vtable*) created when C++ compiles a class that has one or more `virtual` functions
    - First level of pointers
    - Contains function pointers to `virtual` functions
    - Used to select the proper function implementation each time a `virtual` function of that class is called
    - If pure `virtual`, function pointer is set to 0
    - Any class that has one or more null pointers in its *vtable* is an abstract class

13.7 (Optional) Polymorphism, Virtual Functions and Dynamic Binding "Under the Hood" (Cont.)

- **How can C++ implement polymorphism, `virtual` functions and dynamic binding internally? (Cont.)**
  - **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**
  - **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**

13.7 (Optional) Polymorphism, Virtual Functions and Dynamic Binding "Under the Hood" (Cont.)

- 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
    - Dereference that pointer to retrieve underlying object
      - Begins with pointer in second level of pointers
    - 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

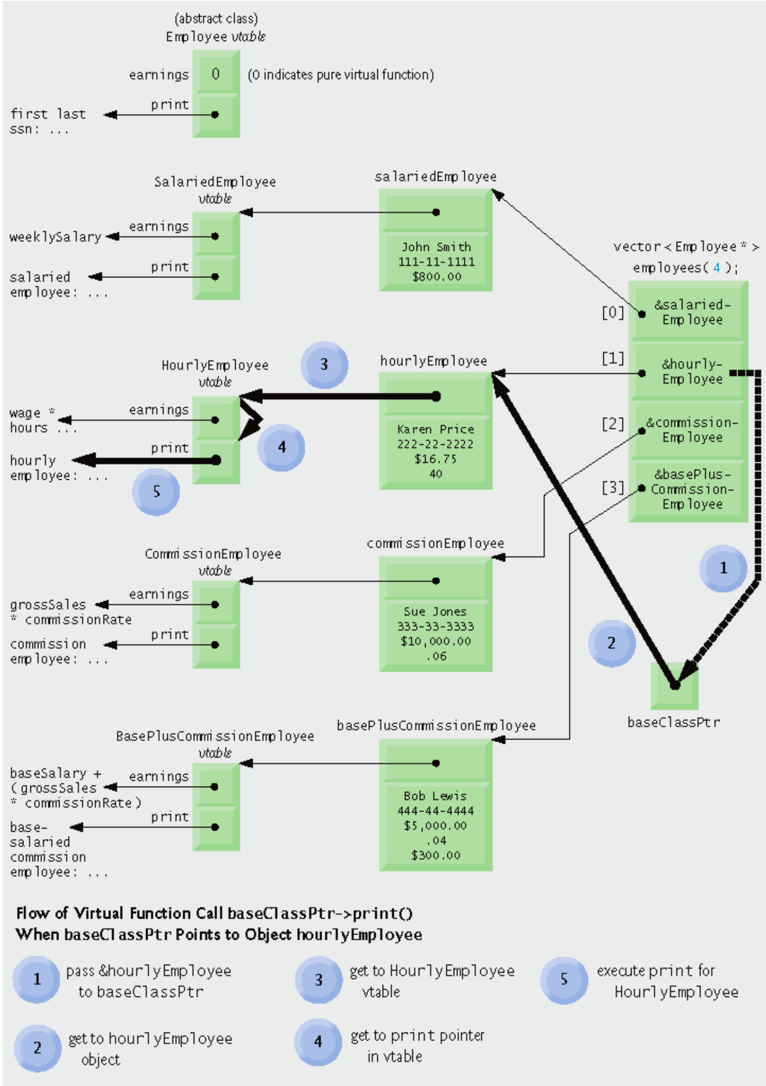

Fig.13.24 | How `virtual` function calls work.

Performance Tip 13.1

Polymorphism, as typically implemented with `virtual` functions and dynamic binding in C++, is efficient. Programmers may use these capabilities with nominal impact on performance.

Performance Tip 13.2

Virtual functions and dynamic binding enable polymorphic programming as an alternative to `switch` logic programming. Optimizing compilers normally generate polymorphic code that runs as efficiently as hand-coded `switch`-based logic. The overhead of polymorphism is acceptable for most applications. But in some situations—real-time applications with stringent performance requirements, for example—the overhead of polymorphism may be too high.

Software Engineering Observation 13.11

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—no source code needs to be revealed. 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 (via dynamic binding).

13.8 Case Study: Payroll System Using Polymorphism and Run-Time Type Information with Downcasting, `dynamic_cast`, `typeid` and `type_info`

- Example: Reward `BasePlusCommissionEmployee`s 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

13.8 Case Study: Payroll System Using Polymorphism and Run-Time Type Information with Downcasting, `dynamic_cast`, `typeid` and `type_info` (Cont.)

- `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

13.8 Case Study: Payroll System Using Polymorphism and Run-Time Type Information with Downcasting, dynamic_cast, typeid and type_info (Cont.)

- 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>

Outline

```
// Fig. 13.25: fig13_25.cpp
// Demonstrating downcasting and run-time type information.
// NOTE: For this example to run in Visual C++ .NET,
// you need to enable RTTI (Run-Time Type Info) for the project.
#include <iostream>
using std::cout;
using std::endl;
using std::fixed;

#include <iomanip>
using std::setprecision;

#include <vector>
using std::vector;

#include <typeinfo>

// include definitions of classes in Employee hierarchy
#include "Employee.h"
#include "SalariedEmployee.h"
#include "HourlyEmployee.h"
#include "CommissionEmployee.h"
#include "BasePlusCommissionEmployee.h"

int main()
{
   // set floating-point output formatting
   cout << fixed << setprecision( 2 );
```

fig13_25.cpp

(1 of 4)

```

   // create vector of four base-class pointers
   vector < Employee * > employees( 4 );

   // initialize vector with various kinds of Employees
   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 );

   // polymorphically process each element in vector employees
   for ( size_t i = 0; i < employees.size(); i++ )
   {
      employees[ i ]->print(); // output employee information
      cout << endl;

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

Create employee objects, only one of type **BasePlusCommissionEmployee**

Downcast the **Employee** pointer to a **BasePlusCommissionEmployee** pointer

```
      // determine whether element points to base-salaried
      // commission employee
      if ( derivedPtr != 0 ) // 0 if not a BasePlusCommissionEmployee
      {
         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;
      } // end if

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

   // release objects pointed to by vector's elements
   for ( size_t j = 0; j < employees.size(); j++ )
   {
      // output class name
      cout << "deleting object of "
         << typeid( *employees[ j ] ).name() << endl;

      delete employees[ j ];
   } // end for

   return 0;
} // end main
```

Determine if cast was successful

If cast was successful, modify base salary

Use **`typeid`** and function **`name`** to display object types

Outline

```
salaried employee: John Smith
social security number: 111-11-1111
weekly salary: 800.00
earned $800.00

hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: 16.75; hours worked: 40.00
earned $670.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: 10000.00; commission rate: 0.06
earned $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: 5000.00; commission rate: 0.04; base salary: 300.00
old base salary: $300.00
new base salary with 10% increase is: $330.00
earned $530.00

deleting object of class SalariedEmployee
deleting object of class HourlyEmployee
deleting object of class CommissionEmployee
deleting object of class BasePlusCommissionEmployee
```

fig13_25.cpp

(4 of 4)

13.9 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
  - 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

Good Programming Practice 13.2

If a class has `virtual` functions, provide a `virtual` destructor, even if one is not required for the class. Classes derived from this class may contain destructors that must be called properly.

Common Programming Error 13.5

Constructors cannot be `virtual`. Declaring a constructor `virtual` is a compilation error.

13.10 (Optional) Software Engineering Case Study: Incorporating Inheritance into the ATM System

- **UML model for inheritance**
  - **The generalization relationship**
    - **The base class is a generalization of the derived classes**
    - **The derived classes are specializations of the base class**
  - **Pure `virtual` functions are abstract operations in the UML**
  - **Generalizations and abstract operations are written in italics**
- ***`Transaction`* base class**
  - **Contains the functions and data members `BalanceInquiry`, `Withdrawal` and `Deposit` have in common**
    - ***`execute`* function**
    - **`accountNumber` data member**

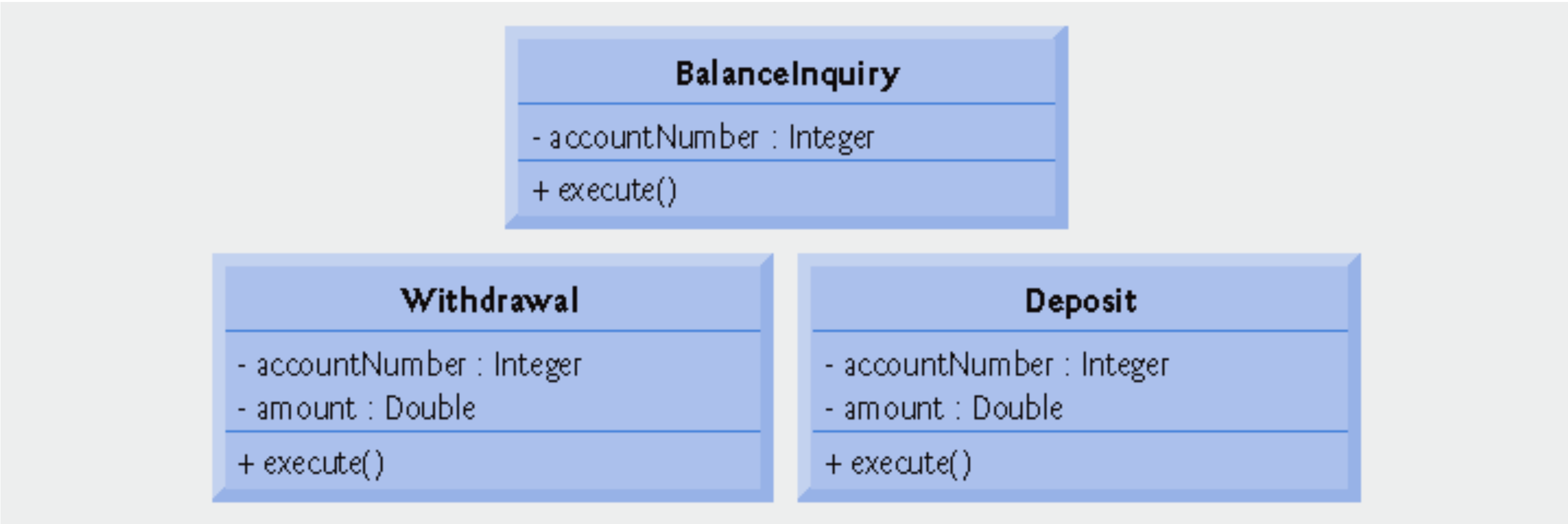

Fig.13.26 | Attributes and operations of classes `BalanceInquiry`, `Withdrawal` and `Deposit`.

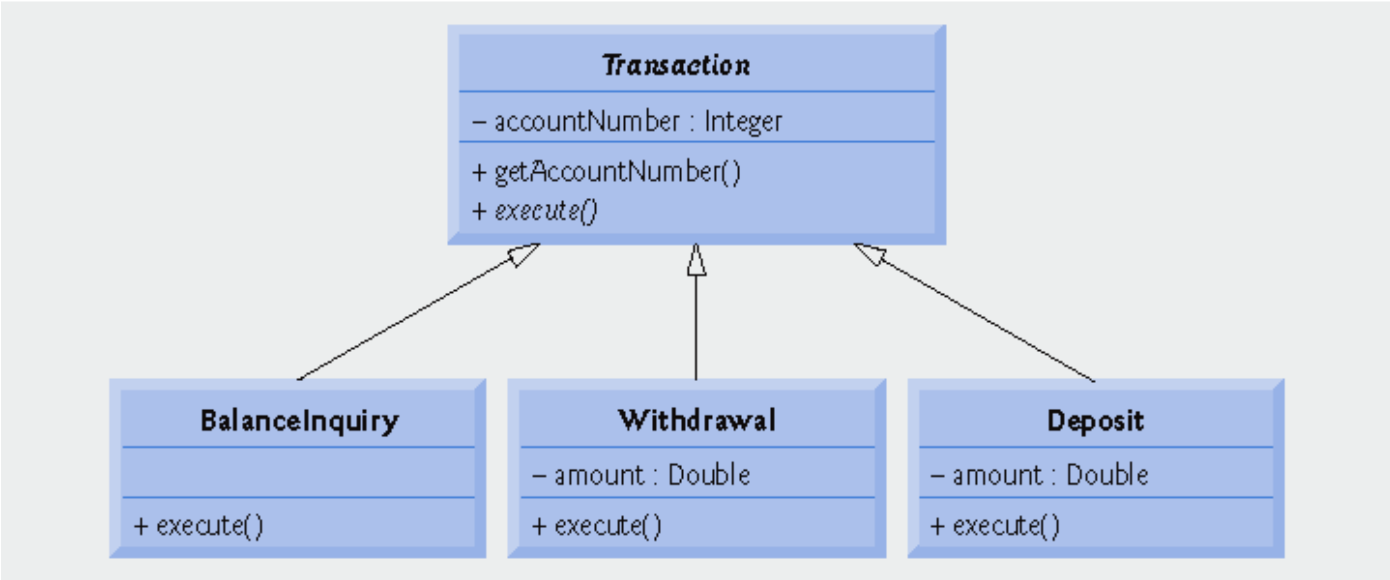

Fig.13.27 | Class diagram modeling generalization relationship between base class `Transaction` and derived classes `BalanceInquiry`, `Withdrawal` and `Deposit`.

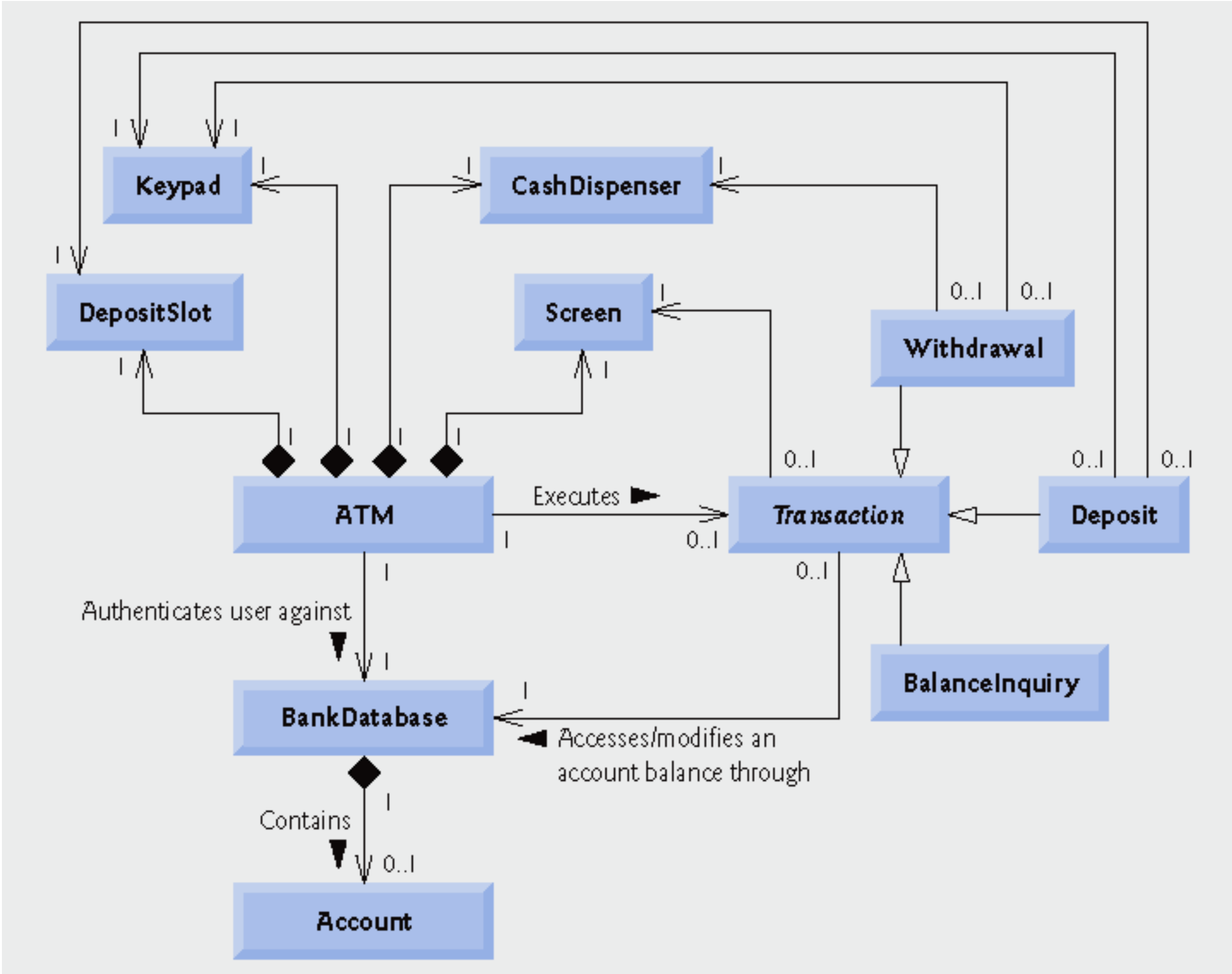

Fig.13.28 | Class diagram of the ATM system (incorporating inheritance). Note that abstract class name `Transaction` appears in italics.

13.10 (Optional) Software Engineering Case Study: Incorporating Inheritance into the ATM System (Cont.)

- **Incorporating inheritance into the ATM system design**
  - **If class `A` is a generalization of class `B`, then class `B` is derived from class `A`**
  - **If class `A` is an abstract class and class `B` is a derived class of class `A`, then class `B` must implement the pure `virtual` functions of class `A` if class `B` is to be a concrete class**

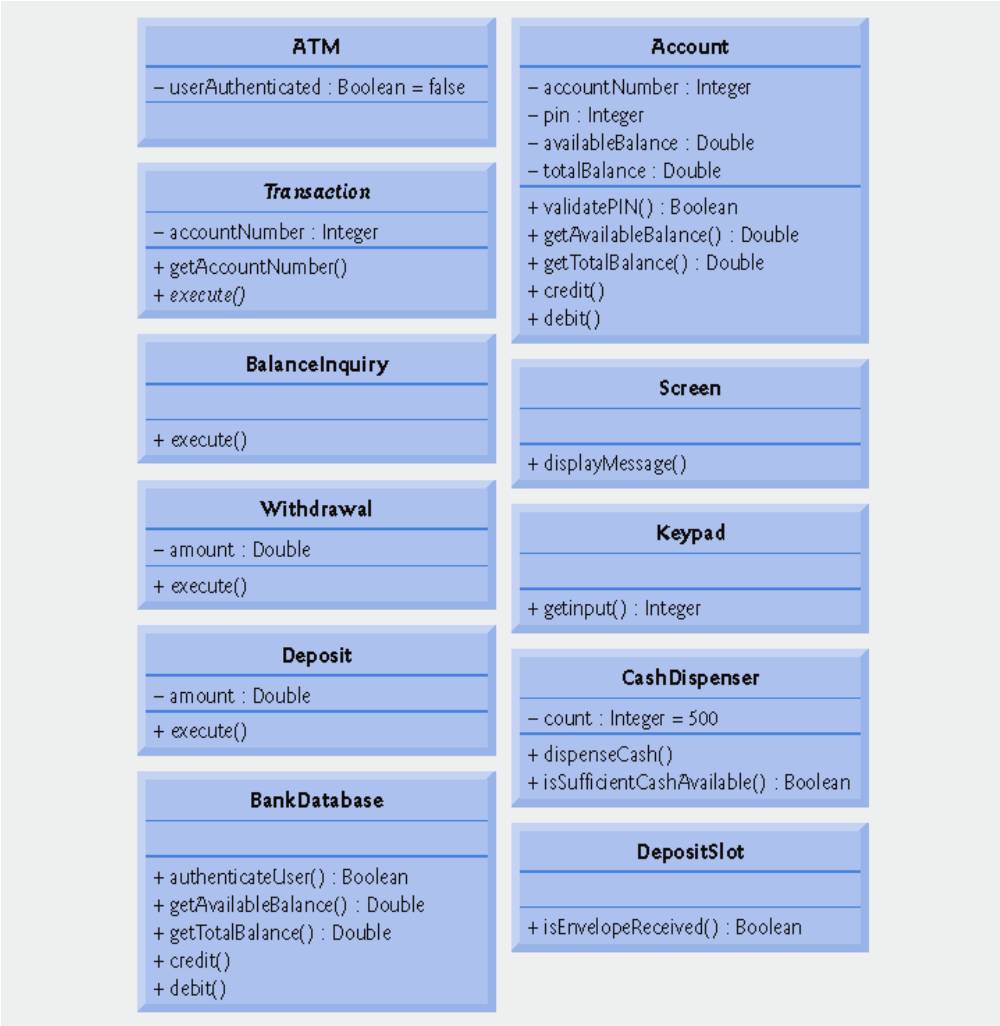

Fig.13.29 | **Class diagram after incorporating inheritance into the system.**

Software Engineering Observation 13.12

A complete class diagram shows all the associations among classes and all the attributes and operations for each class. When the number of class attributes, operations and associations is substantial (as in Fig. 13.28 and Fig. 13.29), a good practice that promotes readability is to divide this information between two class diagrams—one focusing on associations and the other on attributes and operations. However, when examining classes modeled in this fashion, it is crucial to consider both class diagrams to get a complete view of the classes. For example, one must refer to Fig. 13.28 to observe the inheritance relationship between `Transaction` and its derived classes that is omitted from Fig. 13.29.

```
// Fig. 13.30: Withdrawal.h
// Definition of class Withdrawal that represents a withdrawal transaction
#ifndef WITHDRAWAL_H
#define WITHDRAWAL_H

#include "Transaction.h" // Transaction class definition

// class Withdrawal derives from base class Transaction
class Withdrawal : public Transaction
{
}; // end class Withdrawal

#endif // WITHDRAWAL_H
```

Class **`Withdrawal`** inherits from **`Transaction`**

Outline

`Withdrawal.h`

(1 of 1)

```
// Fig. 13.31: Withdrawal.h
// Definition of class Withdrawal that represents a withdrawal transaction
#ifndef WITHDRAWAL_H
#define WITHDRAWAL_H

#include "Transaction.h" // Transaction class definition

class Keypad; // forward declaration of class Keypad
class CashDispenser; // forward declaration of class CashDispenser

// class Withdrawal derives from base class Transaction
class Withdrawal : public Transaction
{
public:
   // member function overriding execute in base class Transaction
   virtual void execute(); // perform the transaction
private:
   // attributes
   double amount; // amount to withdraw
   Keypad &keypad; // reference to ATM's keypad
   CashDispenser &cashDispenser; // reference to ATM's cash dispenser
}; // end class Withdrawal

#endif // WITHDRAWAL_H
```

Class **`Withdrawal`** inherits from **`Transaction`**

`Withdrawal.h`

```
// Fig. 13.32: Transaction.h
// Transaction abstract base class definition.
#ifndef TRANSACTION_H
#define TRANSACTION_H

class Screen; // forward declaration of class Screen
class BankDatabase; // forward declaration of class BankDatabase

class Transaction
{
public:
   int getAccountNumber(); // return account number
   Screen &getScreen(); // return reference to screen
   BankDatabase &getBankDatabase(); // return reference to bank database

   // pure virtual function to perform the transaction
   virtual void execute() = 0; // overridden in derived classes
private:
   int accountNumber; // indicates account involved
   Screen &screen; // reference to the screen of the ATM
   BankDatabase &bankDatabase; // reference to the account info database
}; // end class Transaction

#endif // TRANSACTION_H
```

Transaction is an abstract class, contains a pure **virtual** function

Declare pure **virtual** function **execute**