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

Outline

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: `SpaceObjects`**
  - **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

Commission Employee.h

(1 of 2)

```
1  // Fig. 13.1: CommissionEmployee.h
2  // CommissionEmployee class definition represents a commission employee.
3  #ifndef COMMISSION_H
4  #define COMMISSION_H
5
6  #include <string> // C++ standard string class
7  using std::string;
8
9  class CommissionEmployee
10 {
11 public:
12    CommissionEmployee( const string &, const string &, const string &,
13       double = 0.0, double = 0.0 );
14
15    void setFirstName( const string & ); // set first name
16    string getFirstName() const; // return first name
17
18    void setLastName( const string & ); // set last name
19    string getLastName() const; // return last name
20
21    void setSocialSecurityNumber( const string & ); // set SSN
22    string getSocialSecurityNumber() const; // return SSN
23
24    void setGrossSales( double ); // set gross sales amount
25    double getGrossSales() const; // return gross sales amount
```

```
26
27    void setCommissionRate( double ); // set commission rate
28    double getCommissionRate() const; // return commission rate
29
30    double earnings() const; // calculate earnings
31    void print() const; // print CommissionEmployee object
32 private:
33    string firstName;
34    string lastName;
35    string socialSecurityNumber;
36    double grossSales; // gross weekly sales
37    double commissionRate; // commission percentage
38 }; // end class CommissionEmployee
39
40 #endif
```

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

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

Outline

Employee.h

(2 of 2)

Outline

CommissionEmployee.cpp

(1 of 4)

```
1  // Fig. 13.2: CommissionEmployee.cpp
2  // Class CommissionEmployee member-function definitions.
3  #include <iostream>
4  using std::cout;
5
6  #include "CommissionEmployee.h" // CommissionEmployee class definition
7
8  // constructor
9  CommissionEmployee::CommissionEmployee(
10    const string &first, const string &last, const string &ssn,
11    double sales, double rate )
12    : firstName( first ), lastName( last ), socialSecurityNumber( ssn )
13 {
14    setGrossSales( sales ); // validate and store gross sales
15    setCommissionRate( rate ); // validate and store commission rate
16 } // end CommissionEmployee constructor
17
18 // set first name
19 void CommissionEmployee::setFirstName( const string &first )
20 {
21    firstName = first; // should validate
22 } // end function setFirstName
23
24 // return first name
25 string CommissionEmployee::getFirstName() const
26 {
27    return firstName;
28 } // end function getFirstName
```

Outline

```
29
30 // set last name
31 void CommissionEmployee::setLastName( const string &last )
32 {
33    lastName = last;  // should validate
34 } // end function setLastName
35
36 // return last name
37 string CommissionEmployee::getLastName() const
38 {
39    return lastName;
40 } // end function getLastName
41
42 // set social security number
43 void CommissionEmployee::setSocialSecurityNumber( const string &ssn )
44 {
45    socialSecurityNumber = ssn; // should validate
46 } // end function setSocialSecurityNumber
47
48 // return social security number
49 string CommissionEmployee::getSocialSecurityNumber() const
50 {
51    return socialSecurityNumber;
52 } // end function getSocialSecurityNumber
53
54 // set gross sales amount
55 void CommissionEmployee::setGrossSales( double sales )
56 {
57    grossSales = ( sales < 0.0 ) ? 0.0 : sales;
58 } // end function setGrossSales
```

CommissionEmployee.cpp

(2 of 4)

Outline

Commission Employee.cpp

(3 of 4)

```
59
60 // return gross sales amount
61 double CommissionEmployee::getGrossSales() const
62 {
63    return grossSales;
64 } // end function getGrossSales
65
66 // set commission rate
67 void CommissionEmployee::setCommissionRate( double rate )
68 {
69    commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
70 } // end function setCommissionRate
71
72 // return commission rate
73 double CommissionEmployee::getCommissionRate() const
74 {
75    return commissionRate;
76 } // end function getCommissionRate
77
78 // calculate earnings
79 double CommissionEmployee::earnings() const
80 {
81    return getCommissionRate() * getGrossSales();
82 } // end function earnings
```

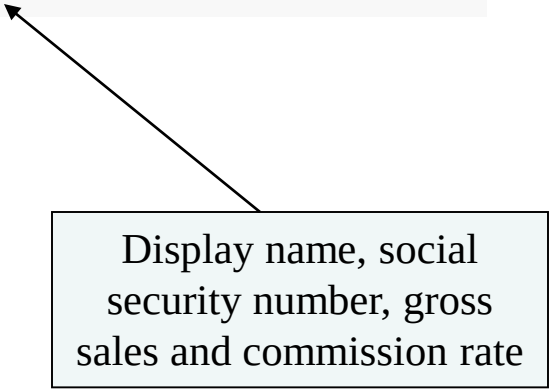
Calculate earnings based on commission rate and gross sales

Outline

```
83
84 // print CommissionEmployee object
85 void CommissionEmployee::print() const
86 {
87    cout << "commission employee: "
88       << getFirstName() << ' ' << getLastName()
89       << "\nsocial security number: " << getSocialSecurityNumber()
90       << "\ngross sales: " << getGrossSales()
91       << "\ncommission rate: " << getCommissionRate();
92 } // end function print
```

Commission Employee.cpp

(4 of 4)



Outline

BasePlus Commission Employee.h

(1 of 1)

```
1  // Fig. 13.3: BasePlusCommissionEmployee.h
2  // BasePlusCommissionEmployee class derived from class
3  // CommissionEmployee.
4  #ifndef BASEPLUS_H
5  #define BASEPLUS_H
6
7  #include <string> // C++ standard string class
8  using std::string;
9
10 #include "CommissionEmployee.h" // CommissionEmployee class declaration
11
12 class BasePlusCommissionEmployee : public CommissionEmployee
13 {
14 public:
15    BasePlusCommissionEmployee( const string &, const string &,
16       const string &, double = 0.0, double = 0.0, double = 0.0 );
17
18    void setBaseSalary( double ); // set base salary
19    double getBaseSalary() const; // return base salary
20
21    double earnings() const; // calculate earnings
22    void print() const; // print BasePlusCommissionEmployee object
23 private:
24    double baseSalary; // base salary
25 }; // end class BasePlusCommissionEmployee
26
27 #endif
```

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

Outline

BasePlus Commission Employee.cpp

(1 of 2)

```
1  // Fig. 13.4: BasePlusCommissionEmployee.cpp
2  // Class BasePlusCommissionEmployee member-function definitions.
3  #include <iostream>
4  using std::cout;
5
6  // BasePlusCommissionEmployee class definition
7  #include "BasePlusCommissionEmployee.h"
8
9  // constructor
10 BasePlusCommissionEmployee::BasePlusCommissionEmployee(
11    const string &first, const string &last, const string &ssn,
12    double sales, double rate, double salary )
13    // explicitly call base-class constructor
14    : CommissionEmployee( first, last, ssn, sales, rate )
15 {
16    setBaseSalary( salary ); // validate and store base salary
17 } // end BasePlusCommissionEmployee constructor
18
19 // set base salary
20 void BasePlusCommissionEmployee::setBaseSalary( double salary )
21 {
22    baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
23 } // end function setBaseSalary
24
25 // return base salary
26 double BasePlusCommissionEmployee::getBaseSalary() const
27 {
28    return baseSalary;
29 } // end function getBaseSalary
```

Outline

```
30
31 // calculate earnings
32 double BasePlusCommissionEmployee::earnings() const
33 {
34    return getBaseSalary() + CommissionEmployee::earnings();
35 } // end function earnings
36
37 // print BasePlusCommissionEmployee object
38 void BasePlusCommissionEmployee::print() const
39 {
40    cout << "base-salaried ";
41
42    // invoke CommissionEmployee's print function
43    CommissionEmployee::print();
44
45    cout << "\nbase salary: " << getBaseSalary();
46 } // end function print
```

BasePlus
Commission
.cpp
(2 of 2)

Redefined `earnings` function incorporates base salary

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

Outline

fig13_05.cpp

(1 of 5)

```
1  // Fig. 13.5: fig13_05.cpp
2  // Aiming base-class and derived-class pointers at base-class
3  // and derived-class objects, respectively.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7  using std::fixed;
8
9  #include <iomanip>
10 using std::setprecision;
11
12 // include class definitions
13 #include "CommissionEmployee.h"
14 #include "BasePlusCommissionEmployee.h"
15
16 int main()
17 {
18    // create base-class object
19    CommissionEmployee commissionEmployee(
20       "Sue", "Jones", "222-22-2222", 10000, .06 );
21
22    // create base-class pointer
23    CommissionEmployee *commissionEmployeePtr = 0;
```

Outline

fig13_05.cpp

(2 of 5)

```
24
25    // create derived-class object
26    BasePlusCommissionEmployee basePlusCommissionEmployee(
27       "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );
28
29    // create derived-class pointer
30    BasePlusCommissionEmployee *basePlusCommissionEmployeePtr = 0;
31
32    // set floating-point output formatting
33    cout << fixed << setprecision( 2 );
34
35    // output objects commissionEmployee and basePlusCommissionEmployee
36    cout << "Print base-class and derived-class objects:\n\n";
37    commissionEmployee.print(); // invokes base-class print
38    cout << "\n\n";
39    basePlusCommissionEmployee.print(); // invokes derived-class print
40
41    // aim base-class pointer at base-class object and print
42    commissionEmployeePtr = &commissionEmployee; // perfectly natural
43    cout << "\n\n\nCalling print with base-class pointer to "
44       << "\nbase-class object invokes base-class print function:\n\n";
45    commissionEmployeePtr->print(); // invokes base-cla
```

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

Outline

fig13_05.cpp

(3 of 5)

```
46
47    // aim derived-class pointer at derived-class object and print
48    basePlusCommissionEmployeePtr = &basePlusCommissionEmployee; // natural
49    cout << "\n\n\nCalling print with derived-class pointer to "
50       << "\nderived-class object invokes derived-class "
51       << "print function:\n\n";
52    basePlusCommissionEmployeePtr->print(); // invokes derived-class print
53
54    // aim base-class pointer at derived-class object and print
55    commissionEmployeePtr = &basePlusCommissionEmployee;
56    cout << "\n\n\nCalling print with base-class pointer to "
57       << "derived-class object\ninvokes base-class print "
58       << "function on that derived-class object:\n\n";
59    commissionEmployeePtr->print(); // invokes base-class print
60    cout << endl;
61    return 0;
62 } // 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 base-class functionality

```
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…)

Outline

fig13_05.cpp

(4 of 5)

(…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
```

Outline

fig13_05.cpp

(5 of 5)

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

fig13_06.cpp

(1 of 2)

```
1  // Fig. 13.6: fig13_06.cpp
2  // Aiming a derived-class pointer at a base-class object.
3  #include "CommissionEmployee.h"
4  #include "BasePlusCommissionEmployee.h"
5
6  int main()
7  {
8     CommissionEmployee commissionEmployee(
9        "Sue", "Jones", "222-22-2222", 10000, .06 );
10    BasePlusCommissionEmployee *basePlusCommissionEmployeePtr = 0;
11
12    // aim derived-class pointer at base-class object
13    // Error: a CommissionEmployee is not a BasePlusCommissionEmployee
14    basePlusCommissionEmployeePtr = &commissionEmployee;
15    return 0;
16 } // end main
```

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
```

Outline

`fig13_06.cpp`

(2 of 2)

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)**

Outline

fig13_07.cpp

(1 of 2)

```
1  // Fig. 13.7: fig13_07.cpp
2  // Attempting to invoke derived-class-only member functions
3  // through a base-class pointer.
4  #include "CommissionEmployee.h"
5  #include "BasePlusCommissionEmployee.h"
6
7  int main()
8  {
9     CommissionEmployee *commissionEmployeePtr = 0; // base class
10    BasePlusCommissionEmployee basePlusCommissionEmployee(
11       "Bob", "Lewis", "333-33-3333", 5000, .04, 300 ); // derived class
12
13    // aim base-class pointer at derived-class object
14    commissionEmployeePtr = &basePlusCommissionEmployee;
15
16    // invoke base-class member functions on derived-class
17    // object through base-class pointer
18    string firstName = commissionEmployeePtr->getFirstName();
19    string lastName = commissionEmployeePtr->getLastName();
20    string ssn = commissionEmployeePtr->getSocialSecurityNumber();
21    double grossSales = commissionEmployeePtr->getGrossSales();
22    double commissionRate = commissionEmployeePtr->getCommissionRate();
23
24    // attempt to invoke derived-class-only member functions
25    // on derived-class object through base-class pointer
26    double baseSalary = commissionEmployeePtr->getBaseSalary();
27    commissionEmployeePtr->setBaseSalary( 500 );
28    return 0;
29 } // end main
```

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

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)
```

Outline

`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

Commission Employee.h

(1 of 2)

```
1  // Fig. 13.8: CommissionEmployee.h
2  // CommissionEmployee class definition represents a commission employee.
3  #ifndef COMMISSION_H
4  #define COMMISSION_H
5
6  #include <string> // C++ standard string class
7  using std::string;
8
9  class CommissionEmployee
10 {
11 public:
12    CommissionEmployee( const string &, const string &, const string &,
13       double = 0.0, double = 0.0 );
14
15    void setFirstName( const string & ); // set first name
16    string getFirstName() const; // return first name
17
18    void setLastName( const string & ); // set last name
19    string getLastName() const; // return last name
20
21    void setSocialSecurityNumber( const string & ); // set SSN
22    string getSocialSecurityNumber() const; // return SSN
23
24    void setGrossSales( double ); // set gross sales amount
25    double getGrossSales() const; // return gross sales amount
```

Outline

```
26
27    void setCommissionRate( double ); // set commission rate
28    double getCommissionRate() const; // return commission rate
29
30    virtual double earnings() const; // calculate earnings
31    virtual void print() const; // print CommissionEmployee object
32 private:
33    string firstName;
34    string lastName;
35    string socialSecurityNumber;
36    double grossSales; // gross weekly sales
37    double commissionRate; // commission percentage
38 }; // end class CommissionEmployee
39
40 #endif
```

Commission Employee.h

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

Outline

BasePlus Commission Employee.h

(1 of 1)

```
1  // Fig. 13.9: BasePlusCommissionEmployee.h
2  // BasePlusCommissionEmployee class derived from class
3  // CommissionEmployee.
4  #ifndef BASEPLUS_H
5  #define BASEPLUS_H
6
7  #include <string> // C++ standard string class
8  using std::string;
9
10 #include "CommissionEmployee.h" // CommissionEmployee class declaration
11
12 class BasePlusCommissionEmployee : public CommissionEmployee
13 {
14 public:
15    BasePlusCommissionEmployee( const string &, const string &,
16       const string &, double = 0.0, double = 0.0, double = 0.0 );
17
18    void setBaseSalary( double ); // set base salary
19    double getBaseSalary() const; // return base salary
20
21    virtual double earnings() const; // calculate earnings
22    virtual void print() const; // print BasePlusCommissionEmployee object
23 private:
24    double baseSalary; // base salary
25 }; // end class BasePlusCommissionEmployee
26
27 #endif
```

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

Outline

fig13_10.cpp

(1 of 5)

```
1  // Fig. 13.10: fig13_10.cpp
2  // Introducing polymorphism, virtual functions and dynamic binding.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6  using std::fixed;
7
8  #include <iomanip>
9  using std::setprecision;
10
11 // include class definitions
12 #include "CommissionEmployee.h"
13 #include "BasePlusCommissionEmployee.h"
14
15 int main()
16 {
17    // create base-class object
18    CommissionEmployee commissionEmployee(
19       "Sue", "Jones", "222-22-2222", 10000, .06 );
20
21    // create base-class pointer
22    CommissionEmployee *commissionEmployeePtr = 0;
23
24    // create derived-class object
25    BasePlusCommissionEmployee basePlusCommissionEmployee(
26       "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );
27
28    // create derived-class pointer
29    BasePlusCommissionEmployee *basePlusCommissionEmployeePtr = 0;
```

Outline

fig13_10.cpp

(2 of 5)

```
30
31    // set floating-point output formatting
32    cout << fixed << setprecision( 2 );
33
34    // output objects using static binding
35    cout << "Invoking print function on base-class and derived-class "
36       << "\nobjects with static binding\n\n";
37    commissionEmployee.print(); // static binding
38    cout << "\n\n";
39    basePlusCommissionEmployee.print(); // static binding
40
41    // output objects using dynamic binding
42    cout << "\n\n\nInvoking print function on base-class and "
43       << "derived-class \nobjects with dynamic binding";
44
45    // aim base-class pointer at base-class object and print
46    commissionEmployeePtr = &commissionEmployee;
47    cout << "\n\nCalling virtual function print with base-class pointer"
48       << "\nto base-class object invokes base-class "
49       << "print function:\n\n";
50    commissionEmployeePtr->print(); // invokes base-class print
```

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

Outline

fig13_10.cpp

(3 of 5)

```
51
52    // aim derived-class pointer at derived-class object and print
53    basePlusCommissionEmployeePtr = &basePlusCommissionEmployee;
54    cout << "\n\nCalling virtual function print with derived-class "
55       << "pointer\nto derived-class object invokes derived-class "
56       << "print function:\n\n";
57    basePlusCommissionEmployeePtr->print(); // invokes derived-class print
58
59    // aim base-class pointer at derived-class object and print
60    commissionEmployeePtr = &basePlusCommissionEmployee;
61    cout << "\n\nCalling virtual function print with base-class poi
62       << "\nto derived-class object invokes derived-class "
63       << "print function:\n\n";
64
65    // polymorphism; invokes BasePlusCommissionEmployee's print;
66    // base-class pointer to derived-class object
67    commissionEmployeePtr->print();
68    cout << endl;
69    return 0;
70 } // 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 …)

Outline

fig13_10.cpp

(4 of 5)

(…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
```

Outline

fig13_10.cpp

(5 of 5)

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).

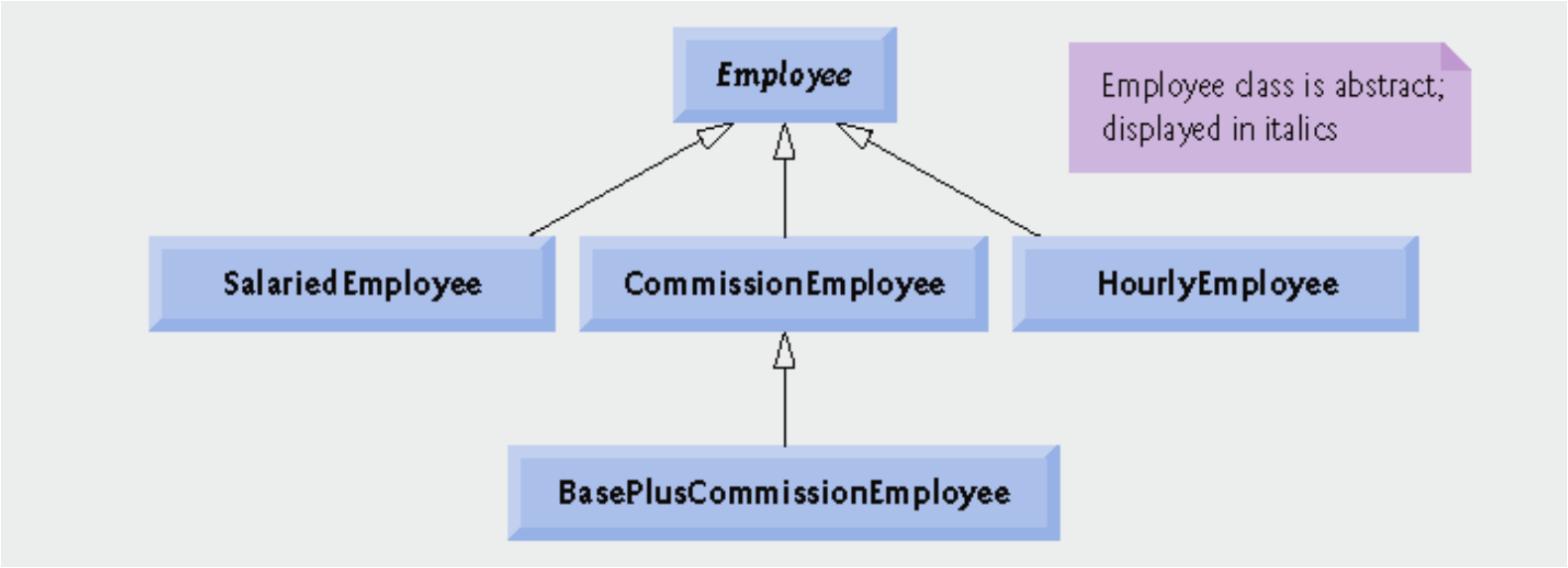


Fig.13.11 | `Employee` hierarchy UML class diagram.

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*

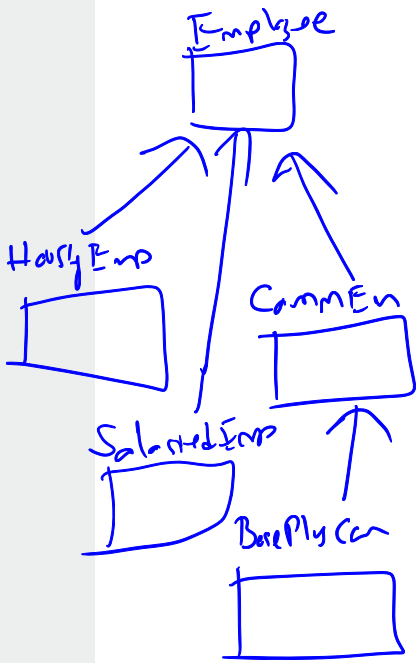


Fig.13.12 | Polymorphic interface for the `Employee` hierarchy classes.

Outline

Employee.h

(1 of 2)

```
1  // Fig. 13.13: Employee.h
2  // Employee abstract base class.
3  #ifndef EMPLOYEE_H
4  #define EMPLOYEE_H
5
6  #include <string> // C++ standard string class
7  using std::string;
8
9  class Employee
10 {
11 public:
12    Employee( const string &, const string &, const string & );
13
14    void setFirstName( const string & ); // set first name
15    string getFirstName() const; // return first name
16
17    void setLastName( const string & ); // set last name
18    string getLastName() const; // return last name
19
20    void setSocialSecurityNumber( const string & ); // set SSN
21    string getSocialSecurityNumber() const; // return SSN
```

Outline

```
22
23    // pure virtual function makes Employee abstract base class
24    virtual double earnings() const = 0; // pure virtual
25    virtual void print() const; // virtual
26 private:
27    string firstName;
28    string lastName;
29    string socialSecurityNumber;
30 }; // end class Employee
31
32 #endif // EMPLOYEE_H
```

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

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

(2 of 2)

Outline

Employee.cpp

(1 of 2)

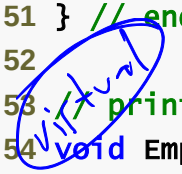
```
1  // Fig. 13.14: Employee.cpp
2  // Abstract-base-class Employee member-function definitions.
3  // Note: No definitions are given for pure virtual functions.
4  #include <iostream>
5  using std::cout;
6
7  #include "Employee.h" // Employee class definition
8
9  // constructor
10 Employee::Employee( const string &first, const string &last,
11    const string &ssn )
12    : firstName( first ), lastName( last ), socialSecurityNumber( ssn )
13 {
14    // empty body
15 } // end Employee constructor
16
17 // set first name
18 void Employee::setFirstName( const string &first )
19 {
20    firstName = first;
21 } // end function setFirstName
22
23 // return first name
24 string Employee::getFirstName() const
25 {
26    return firstName;
27 } // end function getFirstName
28
```

Outline

Employee.cpp

(2 of 2)

```
29 // set last name
30 void Employee::setLastName( const string &last )
31 {
32    lastName = last;
33 } // end function setLastName
34
35 // return last name
36 string Employee::getLastName() const
37 {
38    return lastName;
39 } // end function getLastName
40
41 // set social security number
42 void Employee::setSocialSecurityNumber( const string &ssn )
43 {
44    socialSecurityNumber = ssn; // should validate
45 } // end function setSocialSecurityNumber
46
47 // return social security number
48 string Employee::getSocialSecurityNumber() const
49 {
50    return socialSecurityNumber;
51 } // end function getSocialSecurityNumber
52
53 // print Employee's information (virtual, but not pure virtual)
54 void Employee::print() const
55 {
56    cout << getFirstName() << ' ' << getLastName()
57       << "\nsocial security number: " << getSocialSecurityNumber();
58 } // end function print
```



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

Salaried Employee.h

```
1  // Fig. 13.15: SalariedEmployee.h
2  // SalariedEmployee class derived from Employee.
3  #ifndef SALARIED_H
4  #define SALARIED_H
5
6  #include "Employee.h" // Employee class definition
7
8  class SalariedEmployee : public Employee
9  {
10 public:
11    SalariedEmployee( const string &, const string &,
12       const string &, double = 0.0 );
13
14    void setWeeklySalary( double ); // set weekly salary
15    double getWeeklySalary() const; // return weekly salary
16
17    // keyword virtual signals intent to override
18    virtual double earnings() const; // calculate earnings
19    virtual void print() const; // print SalariedEmployee object
20 private:
21    double weeklySalary; // salary per week
22 }; // end class SalariedEmployee
23
24 #endif // SALARIED_H
```

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

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

Outline

Salaried Employee.cpp

(1 of 2)

```
1  // Fig. 13.16: SalariedEmployee.cpp
2  // SalariedEmployee class member-function definitions.
3  #include <iostream>
4  using std::cout;
5
6  #include "SalariedEmployee.h" // SalariedEmployee class definition
7
8  // constructor
9  SalariedEmployee::SalariedEmployee( const string &first,
10    const string &last, const string &ssn, double salary )
11    : Employee( first, last, ssn )
12 {
13    setWeeklySalary( salary );
14 } // end SalariedEmployee constructor
15
16 // set salary
17 void SalariedEmployee::setWeeklySalary( double salary )
18 {
19    weeklySalary = ( salary < 0.0 ) ? 0.0 : salary;
20 } // end function setWeeklySalary
21
22 // return salary
23 double SalariedEmployee::getWeeklySalary() const
24 {
25    return weeklySalary;
26 } // end function getWeeklySalary
```

Maintain new data member **weeklySalary**

Outline

```
27
28 // calculate earnings;
29 // override pure virtual function earnings in Employee
30 double SalariedEmployee::earnings() const
31 {
32    return getWeeklySalary();
33 } // end function earnings
34
35 // print SalariedEmployee's information
36 void SalariedEmployee::print() const
37 {
38    cout << "salaried employee: ";
39    Employee::print(); // reuse abstract base-class print function
40    cout << "\nweekly salary: " << getWeeklySalary();
41 } // end function print
```

SalariedEmployee.cpp

(1 of 2)

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

```
1  // Fig. 13.17: HourlyEmployee.h
2  // HourlyEmployee class definition.
3  #ifndef HOURLY_H
4  #define HOURLY_H
5
6  #include "Employee.h" // Employee class definition
7
8  class HourlyEmployee : public Employee
9  {
10 public:
11    HourlyEmployee( const string &, const string &,
12       const string &, double = 0.0, double = 0.0 );
13
14    void setWage( double ); // set hourly wage
15    double getWage() const; // return hourly wage
16
17    void setHours( double ); // set hours worked
18    double getHours() const; // return hours worked
19
20    // keyword virtual signals intent to override
21    virtual double earnings() const; // calculate earnings
22    virtual void print() const; // print HourlyEmployee object
23 private:
24    double wage; // wage per hour
25    double hours; // hours worked for week
26 }; // end class HourlyEmployee
27
28 #endif // HOURLY_H
```

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

Functions will be overridden (or defined for first time)

1 2 3 4 5

6

override

Outline

Hourly Employee.cpp

(1 of 2)

```
1  // Fig. 13.18: HourlyEmployee.cpp
2  // HourlyEmployee class member-function definitions.
3  #include <iostream>
4  using std::cout;
5
6  #include "HourlyEmployee.h" // HourlyEmployee class definition
7
8  // constructor
9  HourlyEmployee::HourlyEmployee( const string &first, const string &last,
10    const string &ssn, double hourlyWage, double hoursWorked )
11    : Employee( first, last, ssn )
12 {
13    setWage( hourlyWage ); // validate hourly wage
14    setHours( hoursWorked ); // validate hours worked
15 } // end HourlyEmployee constructor
16
17 // set wage
18 void HourlyEmployee::setWage( double hourlyWage )
19 {
20    wage = ( hourlyWage < 0.0 ? 0.0 : hourlyWage );
21 } // end function setWage
22
23 // return wage
24 double HourlyEmployee::getWage() const
25 {
26    return wage;
27 } // end function getWage
```

Maintain new data member, **hourlyWage**

Outline

```
28
29 // set hours worked
30 void HourlyEmployee::setHours( double hoursWorked )
31 {
32    hours = ( ( ( hoursWorked >= 0.0 ) && ( hoursWorked <= 168.0 ) ) ?
33       hoursWorked : 0.0 );
34 } // end function setHours
35
36 // return hours worked
37 double HourlyEmployee::getHours() const
38 {
39    return hours;
40 } // end function getHours
41
42 // calculate earnings;
43 // override pure virtual function earnings in Employee
44 double HourlyEmployee::earnings() const
45 {
46    if ( getHours() <= 40 ) // no overtime
47       return getWage() * getHours();
48    else
49       return 40 * getWage() + ( ( getHours() - 40 ) * getWage() * 1.5 );
50 } // end function earnings
51
52 // print HourlyEmployee's information
53 void HourlyEmployee::print() const
54 {
55    cout << "hourly employee: ";
56    Employee::print(); // code reuse
57    cout << "\nhourly wage: " << getWage() <<
58       "; hours worked: " << getHours();
59 } // end function print
```

Maintain new data member, **hoursWorked**

(2 of 2)

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

Commission Employee.h

```
1  // Fig. 13.19: CommissionEmployee.h
2  // CommissionEmployee class derived from Employee.
3  #ifndef COMMISSION_H
4  #define COMMISSION_H
5
6  #include "Employee.h" // Employee class definition
7
8  class CommissionEmployee : public Employee
9  {
10 public:
11    CommissionEmployee( const string &, const string &,
12       const string &, double = 0.0, double = 0.0 );
13
14    void setCommissionRate( double ); // set commission rate
15    double getCommissionRate() const; // return commission rate
16
17    void setGrossSales( double ); // set gross sales amount
18    double getGrossSales() const; // return gross sales amount
19
20    // keyword virtual signals intent to override
21    virtual double earnings() const; // calculate earnings
22    virtual void print() const; // print CommissionEmployee object
23 private:
24    double grossSales; // gross weekly sales
25    double commissionRate; // commission percentage
26 }; // end class CommissionEmployee
27
28 #endif // COMMISSION_H
```

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

Functions will be overridden (or defined for first time)

Outline

Commission Employee.cpp

(1 of 2)

```
1  // Fig. 13.20: CommissionEmployee.cpp
2  // CommissionEmployee class member-function definitions.
3  #include <iostream>
4  using std::cout;
5
6  #include "CommissionEmployee.h" // CommissionEmployee class definition
7
8  // constructor
9  CommissionEmployee::CommissionEmployee( const string &first,
10    const string &last, const string &ssn, double sales, double rate )
11    : Employee( first, last, ssn )
12 {
13    setGrossSales( sales );
14    setCommissionRate( rate );
15 } // end CommissionEmployee constructor
16
17 // set commission rate
18 void CommissionEmployee::setCommissionRate( double rate )
19 {
20    commissionRate = ( ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0 );
21 } // end function setCommissionRate
22
23 // return commission rate
24 double CommissionEmployee::getCommissionRate() const
25 {
26     return commissionRate;
27 } // end function getCommissionRate
```

Maintain new data member, **commissionRate**

Outline

```
28
29 // set gross sales amount
30 void CommissionEmployee::setGrossSales( double sales )
31 {
32    grossSales = ( ( sales < 0.0 ) ? 0.0 : sales );
33 } // end function setGrossSales
34
35 // return gross sales amount
36 double CommissionEmployee::getGrossSales() const
37 {
38     return grossSales;
39 } // end function getGrossSales
40
41 // calculate earnings;
42 // override pure virtual function earnings in Employee
43 double CommissionEmployee::earnings() const
44 {
45    return getCommissionRate() * getGrossSales();
46 } // end function earnings
47
48 // print CommissionEmployee's information
49 void CommissionEmployee::print() const
50 {
51    cout << "commission employee: ";
52    Employee::print(); // code reuse
53    cout << "\ngross sales: " << getGrossSales()
54       << "; commission rate: " << getCommissionRate();
55 } // end function print
```

Maintain new data member, **grossSales**

ssion
yee.cpp

(2 of 2)

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

BasePlus Commission Employee.h

```
1  // Fig. 13.21: BasePlusCommissionEmployee.h
2  // BasePlusCommissionEmployee class derived from Employee.
3  #ifndef BASEPLUS_H
4  #define BASEPLUS_H
5
6  #include "CommissionEmployee.h" // CommissionEmployee class definition
7
8  class BasePlusCommissionEmployee : public CommissionEmployee
9  {
10 public:
11    BasePlusCommissionEmployee( const string &, const string &,
12       const string &, double = 0.0, double = 0.0, double = 0.0 );
13
14    void setBaseSalary( double ); // set base salary
15    double getBaseSalary() const; // return base salary
16
17    // keyword virtual signals intent to override
18    virtual double earnings() const; // calculate earnings
19    virtual void print() const; // print BasePlusCommissionEmployee object
20 private:
21    double baseSalary; // base salary per week
22 }; // end class BasePlusCommissionEmployee
23
24 #endif // BASEPLUS_H
```

BasePlusCommissionEmployee inherits from **CommissionEmployee**, already concrete

Functions will be overridden

Outline

BasePlus Commission Employee.cpp

(1 of 2)

```
1  // Fig. 13.22: BasePlusCommissionEmployee.cpp
2  // BasePlusCommissionEmployee member-function definitions.
3  #include <iostream>
4  using std::cout;
5
6  // BasePlusCommissionEmployee class definition
7  #include "BasePlusCommissionEmployee.h"
8
9  // constructor
10 BasePlusCommissionEmployee::BasePlusCommissionEmployee(
11    const string &first, const string &last, const string &ssn,
12    double sales, double rate, double salary )
13    : CommissionEmployee( first, last, ssn, sales, rate )
14 {
15    setBaseSalary( salary ); // validate and store base salary
16 } // end BasePlusCommissionEmployee constructor
17
18 // set base salary
19 void BasePlusCommissionEmployee::setBaseSalary( double salary )
20 {
21    baseSalary = ( ( salary < 0.0 ) ? 0.0 : salary );
22 } // end function setBaseSalary
23
24 // return base salary
25 double BasePlusCommissionEmployee::getBaseSalary() const
26 {
27     return baseSalary;
28 } // end function getBaseSalary
```

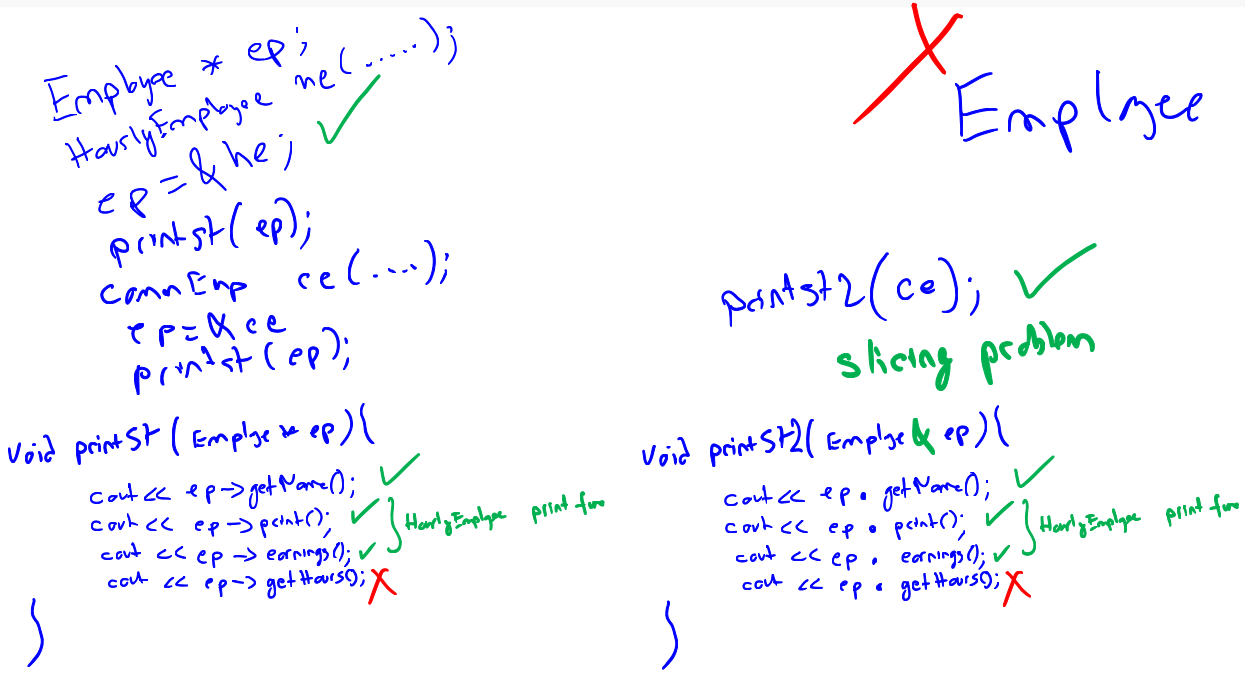
Maintain new data member, **baseSalary**

Outline

BasePlus Commission Employee.cpp

```
29
30 // calculate earnings;
31 // override pure virtual function earnings in Employee
32 double BasePlusCommissionEmployee::earnings() const
33 {
34    return getBaseSalary() + CommissionEmployee::earnings();
35 } // end function earnings
36
37 // print BasePlusCommissionEmployee's information
38 void BasePlusCommissionEmployee::print() const
39 {
40    cout << "base-salaried ";
41    CommissionEmployee::print(); // code reuse
42    cout << "; base salary: " << getBaseSalary();
43 } // end function print
```

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)

```
1  // Fig. 13.23: fig13_23.cpp
2  // Processing Employee derived-class objects individually
3  // and polymorphically using dynamic binding.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7  using std::fixed;
8
9  #include <iomanip>
10 using std::setprecision;
11
12 #include <vector>
13 using std::vector;
14
15 // include definitions of classes in Employee hierarchy
16 #include "Employee.h"
17 #include "SalariedEmployee.h"
18 #include "HourlyEmployee.h"
19 #include "CommissionEmployee.h"
20 #include "BasePlusCommissionEmployee.h"
21
22 void virtualViaPointer( const Employee * const ); // prototype
23 void virtualViaReference( const Employee & ); // prototype
```

f (Emplyee e);

Outline

fig13_23.cpp

(2 of 7)

```
24
25 int main()
26 {
27    // set floating-point output formatting
28    cout << fixed << setprecision( 2 );
29
30    // create derived-class objects
31    SalariedEmployee salariedEmployee(
32       "John", "Smith", "111-11-1111", 800 );
33    HourlyEmployee hourlyEmployee(
34       "Karen", "Price", "222-22-2222", 16.75, 40 );
35    CommissionEmployee commissionEmployee(
36       "Sue", "Jones", "333-33-3333", 10000, .06 );
37    BasePlusCommissionEmployee basePlusCommissionEmployee(
38       "Bob", "Lewis", "444-44-4444", 5000, .04, 300 );
39
40    cout << "Employees processed individually using static binding:\n\n";
41
42    // output each Employee's information and earnings using static binding
43    salariedEmployee.print();
44    cout << "\nearned $" << salariedEmployee.earnings() << "\n\n";
45    hourlyEmployee.print();
46    cout << "\nearned $" << hourlyEmployee.earnings() << "\n\n";
47    commissionEmployee.print();
48    cout << "\nearned $" << commissionEmployee.earnings() << "\n\n";
49    basePlusCommissionEmployee.print();
50    cout << "\nearned $" << basePlusCommissionEmployee.earnings()
51       << "\n\n";
```

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

Outline

```
52
53    // create vector of four base-class pointers
54    vector < Employee * > employees( 4 );
55
56    // initialize vector with Employees
57    employees[ 0 ] = &salariedEmployee;
58    employees[ 1 ] = &hourlyEmployee;
59    employees[ 2 ] = &commissionEmployee;
60    employees[ 3 ] = &basePlusCommissionEmployee;
61
62    cout << "Employees processed polymorphically via dynamic binding:\n\n";
63
64    // call virtualViaPointer to print each Employee's information
65    // and earnings using dynamic binding
66    cout << "Virtual function calls made off base-class pointers:\n\n";
67    virtualViaPointer( employees[0] );
68    for ( size_t i = 0; i < employees.size(); i++ )
69       virtualViaPointer( employees[ i ] );
70
71    // call virtualViaReference to print each Employee's information
72    // and earnings using dynamic binding
73    cout << "Virtual function calls made off base-class references:\n\n";
74
75    for ( size_t i = 0; i < employees.size(); i++ )
76       virtualViaReference( *employees[ i ] ); // note dereferencing
77
78    return 0;
79 } // end main
```

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

Demonstrate dynamic binding using first pointers, then references

Outline

```
80
81 // call Employee virtual functions print and earnings off a
82 // base-class pointer using dynamic binding
83 void virtualViaPointer( const Employee * const baseClassPtr )
84 {
85    baseClassPtr->print();
86    cout << "\nearned $" << baseClassPtr->earnings() << "\n\n";
87 } // end function virtualViaPointer
88
89 // call Employee virtual functions print and earnings off a
90 // base-class reference using dynamic binding
91 void virtualViaReference( const Employee &baseClassRef )
92 {
93    baseClassRef.print();
94    cout << "\nearned $" << baseClassRef.earnings() << "\n\n";
95 } // end function virtualViaReference
```

fig13_23.cpp

(4 of 7)

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...)

Outline

fig13_23.cpp

(5 of 7)

(...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...)

Outline

`fig13_23.cpp`

(6 of 7)

(...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
```

Outline

fig13_23.cpp

(7 of 7)

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

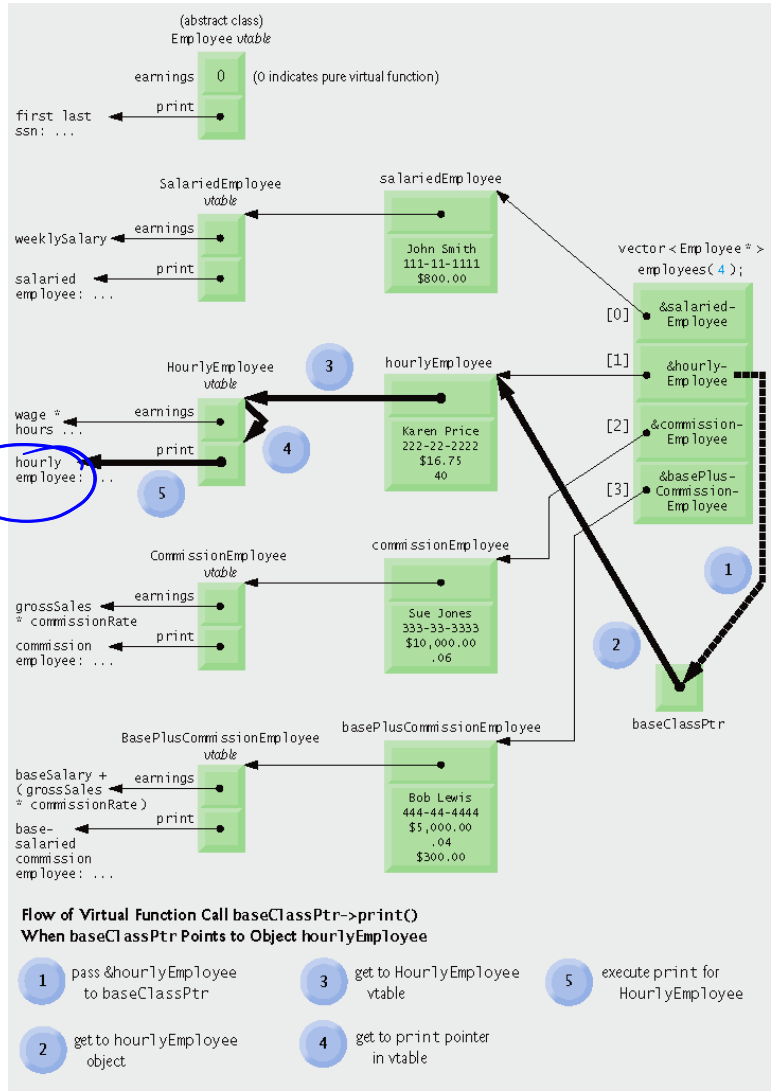
- **How can C++ implement polymorphism, `virtual` functions and dynamic binding internally?**
  - **Three levels of pointers ("triple indirection")**
  - **Virtual function table (*vtable*) created when C++ compiles a class that has one or more `virtual` functions**
    - **First level of pointers**
    - **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



vtable
virtual function table

employees[1]->print();

class A{};

A a;

sizeof(a);

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

fig13_25.cpp

(1 of 4)

```
1  // Fig. 13.25: fig13_25.cpp
2  // Demonstrating downcasting and run-time type information.
3  // NOTE: For this example to run in Visual C++ .NET,
4  // you need to enable RTTI (Run-Time Type Info) for the project.
5  #include <iostream>
6  using std::cout;
7  using std::endl;
8  using std::fixed;
9
10 #include <iomanip>
11 using std::setprecision;
12
13 #include <vector>
14 using std::vector;
15
16 #include <typeinfo>
17
18 // include definitions of classes in Employee hierarchy
19 #include "Employee.h"
20 #include "SalariedEmployee.h"
21 #include "HourlyEmployee.h"
22 #include "CommissionEmployee.h"
23 #include "BasePlusCommissionEmployee.h"
24
25 int main()
26 {
27    // set floating-point output formatting
28    cout << fixed << setprecision( 2 );
```

Outline

fig13_25.cpp

(2 of 4)

```
29
30    // create vector of four base-class pointers
31    vector < Employee * > employees( 4 );
32
33    // initialize vector with various kinds of Employees
34    employees[ 0 ] = new SalariedEmployee(
35       "John", "Smith", "111-11-1111", 800 );
36    employees[ 1 ] = new HourlyEmployee(
37       "Karen", "Price", "222-22-2222", 16.75, 40 );
38    employees[ 2 ] = new CommissionEmployee(
39       "Sue", "Jones", "333-33-3333", 10000, .06 );
40    employees[ 3 ] = new BasePlusCommissionEmployee(
41       "Bob", "Lewis", "444-44-4444", 5000, .04, 300 );
42
43    // polymorphically process each element in vector employees
44    for ( size_t i = 0; i < employees.size(); i++ )
45    {
46       employees[ i ]->print(); // output employee information
47       cout << endl;
48
49       // downcast pointer
50       BasePlusCommissionEmployee *derivedPtr =
51          dynamic_cast < BasePlusCommissionEmployee * >
52             ( employees[ i ] );
```

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

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

Outline

```
53
54    // determine whether element points to base-salaried
55    // commission employee
56    if ( derivedPtr != 0 ) // 0 if not a BasePlusCommissionEmployee
57    {
58       double oldBaseSalary = derivedPtr->getBaseSalary();
59       cout << "old base salary: $" << oldBaseSalary << endl;
60       derivedPtr->setBaseSalary( 1.10 * oldBaseSalary );
61       cout << "new base salary with 10% increase is: $"
62          << derivedPtr->getBaseSalary() << endl;
63    } // end if
64
65    cout << "earned $" << employees[ i ]->earnings() << "\n\n";
66 } // end for
67
68 // release objects pointed to by vector's elements
69 for ( size_t j = 0; j < employees.size(); j++ )
70 {
71    // output class name
72    cout << "deleting object of "
73       << typeid( *employees[ j ] ).name() << endl;
74
75    delete employees[ j ];
76 } // end for
77
78 return 0;
79 } // end main
```

Determine if cast was successful

(3 of 4)

If cast was successful, modify base salary

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

```
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
```

Outline

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**

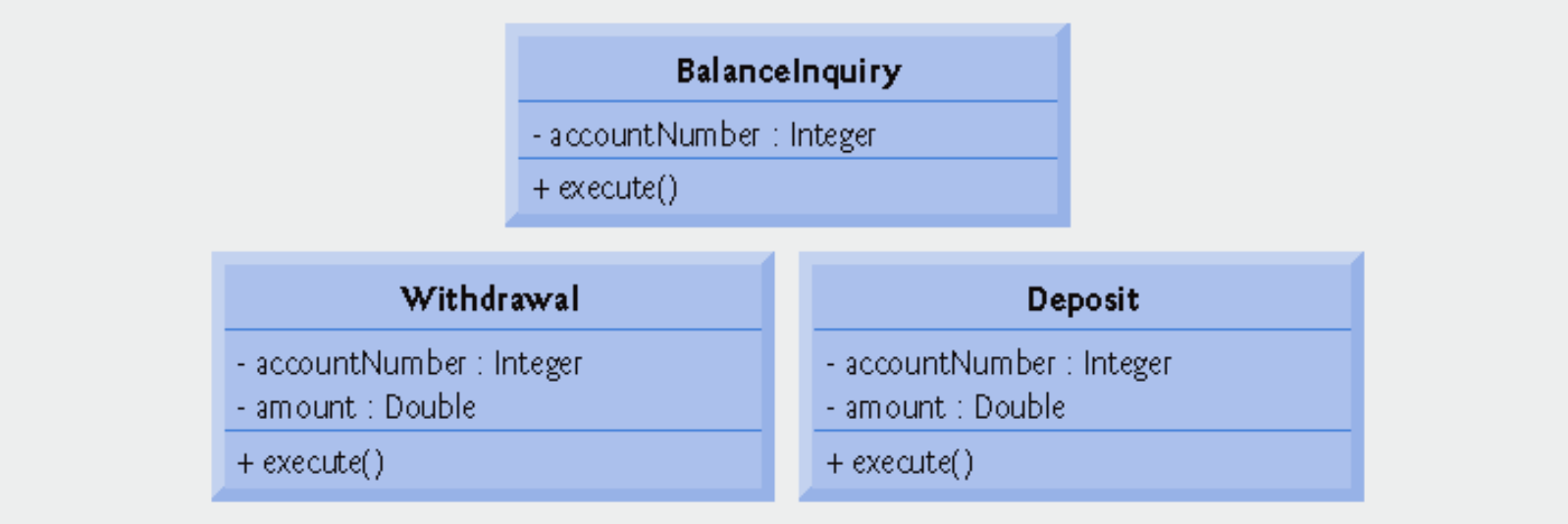


Fig.13.26 | Attributes and operations of classes `BalanceInquiry`, `Withdrawal` and `Deposit`.

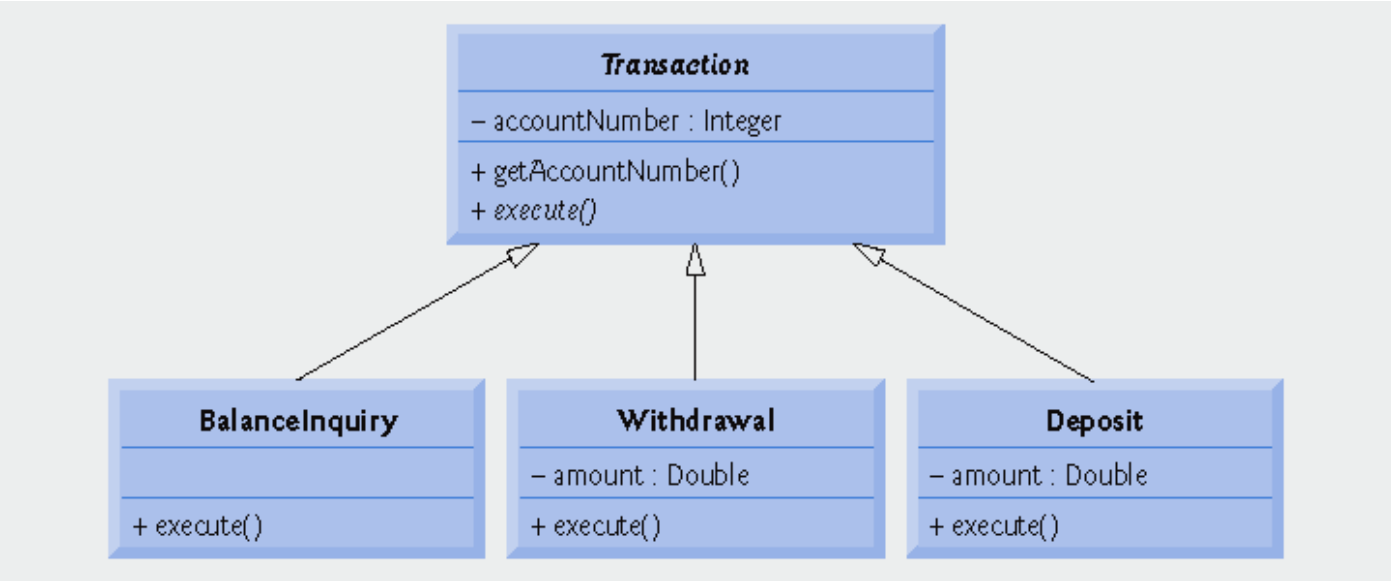


Fig.13.27 | Class diagram modeling generalization relationship between base class `Transaction` and derived classes `BalanceInquiry`, `Withdrawal` and `Deposit`.

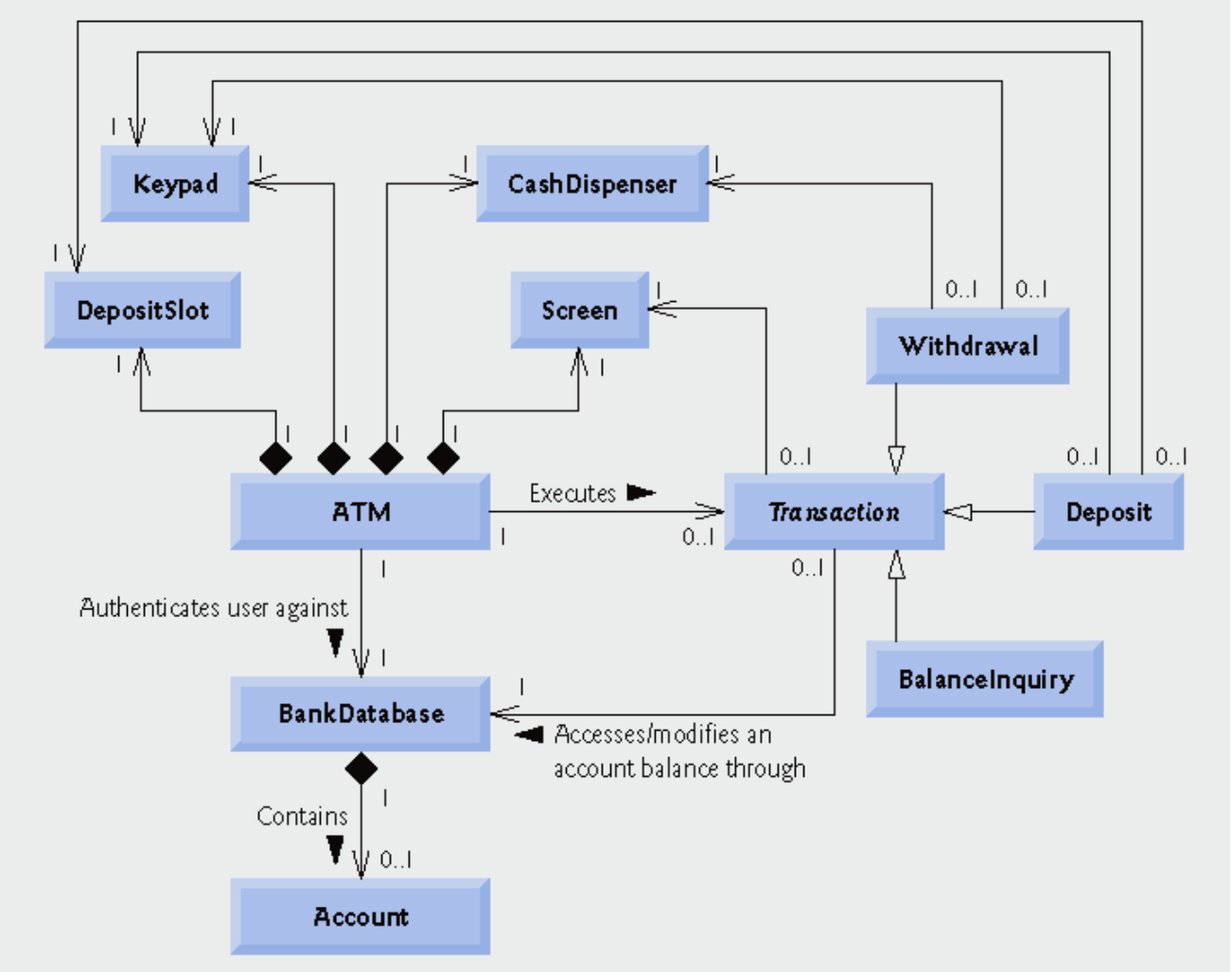


Fig.13.28 | Class diagram of the ATM system (incorporating inheritance). Note that abstract class name `Transaction` appears in italics.

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**

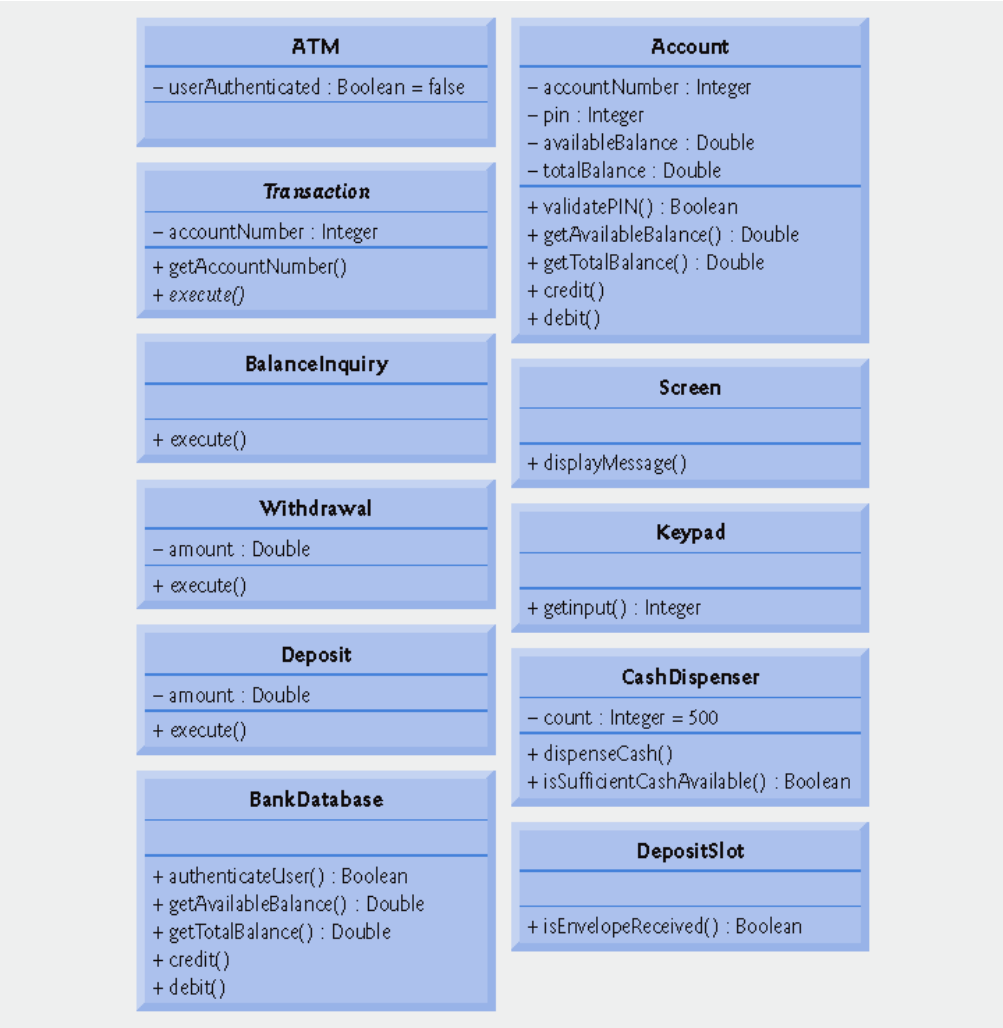


Fig.13.29 | Class diagram after incorporating inheritance into the system.

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.

Outline

Withdrawal.h

(1 of 1)

```
1  // Fig. 13.30: Withdrawal.h
2  // Definition of class Withdrawal that represents a withdrawal transaction
3  #ifndef WITHDRAWAL_H
4  #define WITHDRAWAL_H
5
6  #include "Transaction.h" // Transaction class definition
7
8  // class Withdrawal derives from base class Transaction
9  class Withdrawal : public Transaction
10 {
11 }; // end class Withdrawal
12
13 #endif // WITHDRAWAL_H
```

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

Outline

Withdrawal.h

(1 of 1)

```
1  // Fig. 13.31: Withdrawal.h
2  // Definition of class Withdrawal that represents a withdrawal transaction
3  #ifndef WITHDRAWAL_H
4  #define WITHDRAWAL_H
5
6  #include "Transaction.h" // Transaction class definition
7
8  class Keypad; // forward declaration of class Keypad
9  class CashDispenser; // forward declaration of class CashDispenser
10
11 // class Withdrawal derives from base class Transaction
12 class Withdrawal : public Transaction
13 {
14 public:
15    // member function overriding execute in base class Transaction
16    virtual void execute(); // perform the transaction
17 private:
18    // attributes
19    double amount; // amount to withdraw
20    Keypad &keypad; // reference to ATM's keypad
21    CashDispenser &cashDispenser; // reference to ATM's cash dispenser
22 }; // end class Withdrawal
23
24 #endif // WITHDRAWAL_H
```

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

Outline

Transaction.h

(1 of 1)

```
1  // Fig. 13.32: Transaction.h
2  // Transaction abstract base class definition.
3  #ifndef TRANSACTION_H
4  #define TRANSACTION_H
5
6  class Screen; // forward declaration of class Screen
7  class BankDatabase; // forward declaration of class BankDatabase
8
9  class Transaction
10 {
11 public:
12    int getAccountNumber(); // return account number
13    Screen &getScreen(); // return reference to screen
14    BankDatabase &getBankDatabase(); // return reference to bank database
15
16    // pure virtual function to perform the transaction
17    virtual void execute() = 0; // overridden in derived classes
18 private:
19    int accountNumber; // indicates account involved
20    Screen &screen; // reference to the screen of the ATM
21    BankDatabase &bankDatabase; // reference to the account info database
22 }; // end class Transaction
23
24 #endif // TRANSACTION_H
```

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

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