IS 0020
Program Design and Software Tools

Polymorphism

Lecture 8

September 28/29, 2004

Introduction

- Polymorphism
  - "Program in the general"
  - Derived-class object can be treated as base-class object
    - "is-a" relationship
    - Base class is not a derived class object
  - Virtual functions and dynamic binding
  - Makes programs extensible
    - New classes added easily, can still be processed
- Examples
  - Use abstract base class **`Shape`**
    - Defines common interface (functionality)
    - **`Point`**, **`Circle`** and **`Cylinder`** inherit from **`Shape`**

Invoking Base-Class Functions from Derived-Class Objects

- Pointers to base/derived objects
  - Base pointer aimed at derived object
    - "is a" relationship
      - **`Circle`** "is a" **`Point`**
    - Will invoke base class functions
  - Can cast base-object's address to derived-class pointer
    - Called down-casting
    - Allows derived-class functionality
- Key point
  - Base-pointer can aim at derived-object - but can only call base-class functions
    - Data type of pointer/reference determines functions it can call

Outline

fig10_05.cpp
(1 of 3)

```
1  // Fig. 10.5: fig10_05.cpp
2  // Aiming base-class and derived-class pointers at base-class
3  // and derived-class objects, respectively.
4  #include <iostream>
5
6  using std::cout;
7  using std::endl;
8  using std::fixed;
9
10 #include <iomanip>
11
12 using std::setprecision;
13
14 #include "point.h"   // Point class definition
15 #include "circle.h"  // Circle class definition
16
17 int main()
18 {
19    Point point( 30, 50 );
20    Point *pointPtr = 0;    // base-class pointer
21
22    Circle circle( 120, 89, 2.7 );
23    Circle *circlePtr = 0;  // derived-class pointer
24
```

```
25     // set floating-point numeric forma
26     cout << fixed << setprecision( 2 );
27
28     // output objects point and circle
29     cout << "Print point and circle obj
30          << "\nPoint: ";
31     point.print();   // invokes Point's print
32     cout << "\nCircle: ";
33     circle.print();  // invokes Circle's print
34
35     // aim base-class pointer at base-class object and print
36     pointPtr = &point;
37     cout << "\n\nCalling print with base-class pointer to "
38          << "\nbase-class object invokes base-class print "
39          << "function:\n";
40     pointPtr->print(); // invokes Point's print
41
42     // aim derived-class pointer at derived-class object
43     // and print
44     circlePtr = &circle;
45     cout << "\n\nCalling print with derived-class pointer to "
46          << "\nderived-class object invokes derived-class "
47          << "print function:\n";
48     circlePtr->print();  // invokes Circle's print
49
```

Use objects and pointers to call the **print** function. The pointers and objects are of the same class, so the proper **print** function is called.

Outline

fig10_05.cpp
(2 of 3)


```
50     // aim base-class pointer at derived-class object and print
51     pointPtr = &circle;
52     cout << "\n\nCalling print with base-class pointer to "
53          << "derived-class object\ninvokes base-class print "
54          << "function on that derived-class object:\n";
55     pointPtr->print();  // invokes Point's print
56     cout << endl;
57
58     return 0;
59
60  } // end main
```

Aiming a base-class pointer at a derived object is allowed (the **Circle** "is a" **Point**). However, it calls **Point**'s print function, determined by the pointer type. **virtual** functions allow us to change this.

Outline

fig10_05.cpp
(3 of 3)

```
1  // Fig. 10.6: fig10_06.cpp
2  // Aiming a derived-class pointer at a base-class object.
3  #include "point.h"   // Point class definition
4  #include "circle.h"  // Circle class definition
5
6  int main()
7  {
8     Point point( 30, 50 );
9     Circle *circlePtr = 0;
10
11    // aim derived-class pointer at base-class object
12    circlePtr = &point;  // Error: a Point is not a Circle
13
14    return 0;
15
16 } // end main
```

```
C:\cpphtp4\examples\ch10\fig10_06\Fig10_06.cpp(12) : error C2440:
'=' : cannot convert from 'class Point *' to 'class Circle *'
        Types pointed to are unrelated; conversion requires
        reinterpret_cast, C-style cast or function-style cast
```

Outline

fig10_06.cpp
(1 of 1)

fig10_06.cpp
output (1 of 1)


```
1  // Fig. 10.7: fig10_07.cpp
2  // Attempting to invoke derived-class-only member functions
3  // through a base-class pointer.
4  #include "point.h"   // Point class definition
5  #include "circle.h"  // Circle class definition
6
7  int main()
8  {
9     Point *pointPtr = 0;
10    Circle circle( 120, 89, 2.7 );
11
12    // aim base-class pointer at derived-class object
13    pointPtr = &circle;
14
15    // invoke base-class member functions on derived-class
16    // object through base-class pointer
17    int x = pointPtr->getX();
18    int y = pointPtr->getY();
19    pointPtr->setX( 10 );
20    pointPtr->setY( 10 );
21    pointPtr->print();
22
```

Outline

fig10_07.cpp
(1 of 2)

```
23     // attempt to invoke derived-class-only member functions
24     // on derived-class object through base-class pointer
25     double radius = pointPtr->getRadius();
26     pointPtr->setRadius( 33.33 );
27     double diameter = pointPtr->getDiameter();
28     double circumference = pointPtr->getCircumference();
29     double area = pointPtr->getArea();
30
31     return 0;
32
33  } // end main
```

Outline

fig10_07.cpp
(2 of 2)

These functions are only defined in **`Circle`**. However, **`pointPtr`** is of class **`Point`**.

Virtual Functions

- **`virtual`** functions
  - Object (not pointer) determines function called
- Why useful?
  - Suppose **`Circle`**, **`Triangle`**, **`Rectangle`** derived from **`Shape`**
    - Each has own **`draw`** function
  - To draw any shape
    - Have base class **`Shape`** pointer, call **`draw`**
    - Program determines proper **`draw`** function at run time (dynamically)
    - Treat all shapes generically

Virtual Functions

- Declare **`draw`** as **`virtual`** in base class
  - Override **`draw`** in each derived class
    - Like redefining, but new function must have same signature
  - If function declared **`virtual`**, can only be overridden
    - **`virtual void draw() const;`**
    - Once declared **`virtual`**, **`virtual`** in all derived classes
      - Good practice to explicitly declare **`virtual`**
- Dynamic binding
  - Choose proper function to call at run time
  - Only occurs off pointer handles
    - If function called from object, uses that object's definition

Virtual Functions

- Polymorphism
  - Same message, "print", given to many objects
    - All through a base pointer
  - Message takes on "many forms"
- Summary
  - Base-pointer to base-object, derived-pointer to derived
    - Straightforward
  - Base-pointer to derived object
    - Can only call base-class functions
  - Derived-pointer to base-object
    - Compiler error
    - Allowed if explicit cast made

13

Polymorphism Examples

- Suppose designing video game
  - Base class **`SpaceObject`**
    - Derived **`Martian`**, **`SpaceShip`**, **`LaserBeam`**
    - Base function **`draw`**
  - To refresh screen
    - Screen manager has **`vector`** of base-class pointers to objects
    - Send **`draw`** message to each object
    - Same message has "many forms" of results
  - Easy to add class **`Mercurian`**
    - Inherits from **`SpaceObject`**
    - Provides own definition for **`draw`**
  - Screen manager does not need to change code
    - Calls **`draw`** regardless of object's type
    - **`Mercurian`** objects "plug right in"

14

Type Fields and switch Structures

- One way to determine object's class
  - Give base class an attribute
    - **`shapeType`** in class **`Shape`**
  - Use **`switch`** to call proper **`print`** function
- Many problems
  - May forget to test for case in **`switch`**
  - If add/remove a class, must update **`switch`** structures
    - Time consuming and error prone
- Better to use polymorphism
  - Less branching logic, simpler programs, less debugging

15

Abstract Classes

- Abstract classes
  - Sole purpose: to be a base class (called abstract base classes)
  - Incomplete
    - Derived classes fill in "missing pieces"
  - Cannot make objects from abstract class
    - However, can have pointers and references
- Concrete classes
  - Can instantiate objects
  - Implement all functions they define
  - Provide specifics

16

Abstract Classes

- Abstract classes not required, but helpful
- To make a class abstract
  - Need one or more "pure" virtual functions
    - Declare function with initializer of 0

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

  - Regular virtual functions
    - Have implementations, overriding is optional
  - Pure virtual functions
    - No implementation, must be overridden
  - Abstract classes can have data and concrete functions
    - Required to have one or more pure virtual functions

Case Study: Inheriting Interface and Implementation

- Make abstract base class **Shape**
  - Pure virtual functions (must be implemented)
    - **`getName`**, **`print`**
    - Default implementation does not make sense
  - Virtual functions (may be redefined)
    - **`getArea`**, **`getVolume`**
      - Initially return **`0.0`**
    - If not redefined, uses base class definition
  - Derive classes **`Point`**, **`Circle`**, **`Cylinder`**

Case Study: Inheriting Interface and Implementation

	`getArea`	`getVolume`	`getName`	`print`
`Shape`	`0.0`	`0.0`	`= 0`	`= 0`
`Point`	`0.0`	`0.0`	`"Point"`	`[x,y]`
`Circle`	πr^2	`0.0`	`"Circle"`	`center=[x,y]; radius=r`
`Cylinder`	$2\pi r^2 + 2\pi rh$	$\pi r^2 h$	`"Cylinder"`	`center=[x,y]; radius=r; height=h`

Outline

shape.h (1 of 1)

```
1  // Fig. 10.12: shape.h
2  // Shape abstract-base-class definition.
3  #ifndef SHAPE_H
4  #define SHAPE_H
5
6  #include <string>  // C++ standard string class
7
8  using std::string;
9
10 class Shape {
11
12 public:
13
14    // virtual function that returns shape area
15    virtual double getArea() const;
16
17    // virtual function that returns shape volume
18    virtual double getVolume() const;
19
20    // pure virtual functions; overridden in derived classes
21    virtual string getName() const = 0; // return shape name
22    virtual void print() const = 0;     // output shape
23
24 }; // end class Shape
25
26 #endif
```

Virtual and pure virtual functions.

Outline

shape.cpp (1 of 1)

```
1  // Fig. 10.13: shape.cpp
2  // Shape class member-function definitions.
3  #include <iostream>
4
5  using std::cout;
6
7  #include "shape.h"  // Shape class definition
8
9  // return area of shape; 0.0 by default
10 double getArea() const
11 {
12    return 0.0;
13
14 }  // end function getArea
15
16 // return volume of shape; 0.0 by default
17 double getVolume() const
18 {
19    return 0.0;
20
21 }  // end function getVolume
```

Polymorphism, Virtual Functions and Dynamic Binding "Under the Hood"

- Polymorphism has overhead
  - Not used in STL (Standard Template Library) to optimize performance
- **`virtual`** function table (vtable)
  - Every class with a **`virtual`** function has a vtable
  - For every **`virtual`** function, vtable has pointer to the proper function
  - If derived class has same function as base class
    - Function pointer aims at base-class function

Virtual Destructors

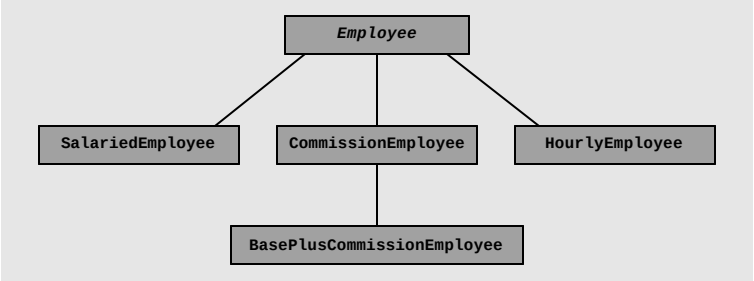
- Base class pointer to derived object
  - If destroyed using **`delete`**, behavior unspecified
- Simple fix
  - Declare base-class destructor virtual
    - Makes derived-class destructors virtual
  - Now, when **`delete`** used appropriate destructor called
- When derived-class object destroyed
  - Derived-class destructor executes first
  - Base-class destructor executes afterwards
- Constructors cannot be virtual

Case Study: Payroll System Using Polymorphism

- Create a payroll program
  - Use virtual functions and polymorphism
- Problem statement
  - 4 types of employees, paid weekly
    - Salaried (fixed salary, no matter the hours)
    - Hourly (overtime [>40 hours] pays time and a half)
    - Commission (paid percentage of sales)
    - Base-plus-commission (base salary + percentage of sales)
      - Boss wants to raise pay by 10%

Payroll System Using Polymorphism

- Base class **`Employee`**
  - Pure virtual function **`earnings`** (returns pay)
    - Pure virtual because need to know employee type
    - Cannot calculate for generic employee
  - Other classes derive from **`Employee`**



Dynamic Cast

- Downcasting
  - **`dynamic_cast`** operator
    - Determine object's type at runtime
    - Returns 0 if not of proper type (cannot be cast)

```
NewClass *ptr = dynamic_cast < NewClass *> objectPtr;
```

- Keyword **`typeid`**
  - Header **`<typeinfo>`**
  - Usage: **`typeid(object)`**
    - Returns **`type_info`** object
    - Has information about type of operand, including name
    - **`typeid(object).name()`**

Outline

employee.h (1 of 2)

```
1  // Fig. 10.23: employee.h
2  // Employee abstract base class.
3  #ifndef EMPLOYEE_H
4  #define EMPLOYEE_H
5
6  #include <string>  // C++ standard string class
7
8  using std::string;
9
10 class Employee {
11
12 public:
13    Employee( const string &, const string &, const string & );
14
15    void setFirstName( const string & );
16    string getFirstName() const;
17
18    void setLastName( const string & );
19    string getLastName() const;
20
21    void setSocialSecurityNumber( const string & );
22    string getSocialSecurityNumber() const;
23
```

```
24    // pure virtual function makes Employee abstract base class
25    virtual double earnings() const = 0;  // pure virtual
26    virtual void print() const;           // virtual
27
28 private:
29    string firstName;
30    string lastName;
31    string socialSecurityNumber;
32
33 }; // end class Employee
34
35 #endif // EMPLOYEE_H
```

Outline

employee.h (2 of 2)


```
1  // Fig. 10.24: employee.cpp
2  // Abstract-base-class Employee member-function definitions.
3  // Note: No definitions are given for pure virtual functions.
4  #include <iostream>
5
6  using std::cout;
7  using std::endl;
8
9  #include "employee.h"  // Employee class definition
10
11 // constructor
12 Employee::Employee( const string &first, const string &last,
13    const string &SSN )
14    : firstName( first ),
15      lastName( last ),
16      socialSecurityNumber( SSN )
17 {
18    // empty body
19
20 } // end Employee constructor
21
```

Outline

employee.cpp
(1 of 3)

Outline

employee.cpp
(2 of 3)

```
22 // return first name
23 string Employee::getFirstName() const
24 {
25    return firstName;
26
27 } // end function getFirstName
28
29 // return last name
30 string Employee::getLastName() const
31 {
32    return lastName;
33
34 } // end function getLastName
35
36 // return social security number
37 string Employee::getSocialSecurityNumber() const
38 {
39    return socialSecurityNumber;
40
41 } // end function getSocialSecurityNumber
42
43 // set first name
44 void Employee::setFirstName( const string &first )
45 {
46    firstName = first;
47
48 } // end function setFirstName
49
```


Outline

employee.cpp
(3 of 3)

```
50 // set last name
51 void Employee::setLastName( const string &last )
52 {
53    lastName = last;
54
55 } // end function setLastName
56
57 // set social security number
58 void Employee::setSocialSecurityNumber( const string &number )
59 {
60    socialSecurityNumber = number;  // should validate
61
62 } // end function setSocialSecurity
63
64 // print Employee's information
65 void Employee::print() const
66 {
67    cout << getFirstName() << ' ' << getLastName()
68         << "\nsocial security number: "
69         << getSocialSecurityNumber() << endl;
70
71 } // end function print
```

Default implementation for virtual function **`print`**.

Outline

salaried.h (1 of 1)

```
1  // Fig. 10.25: salaried.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
12       const string &, double = 0.0 );
13
14    void setWeeklySalary( double );
15    double getWeeklySalary() const;
16
17    virtual double earnings() const;
18    virtual void print() const;  // "salaried employee: "
19
20 private:
21    double weeklySalary;
22
23 }; // end class SalariedEmployee
24
25 #endif // SALARIED_H
```

New functions for the **`SalariedEmployee`** class.

`earnings` must be overridden. **`print`** is overridden to specify that this is a salaried employee.

Outline

salaried.cpp
(1 of 2)

```
1  // Fig. 10.26: salaried.cpp
2  // SalariedEmployee class member-function definitions.
3  #include <iostream>
4
5  using std::cout;
6
7  #include "salaried.h" // SalariedEmployee class definition
8
9  // SalariedEmployee constructor
10 SalariedEmployee::SalariedEmployee
11    const string &last, const strin
12    double salary )
13    : Employee( first, last, socialSecurityNumber )
14 {
15    setWeeklySalary( salary );
16
17 } // end SalariedEmployee constructor
18
19 // set salaried employee's salary
20 void SalariedEmployee::setWeeklySalary( double salary )
21 {
22    weeklySalary = salary < 0.0 ? 0.0 : salary;
23
24 } // end function setWeeklySalary
25
```

Use base class constructor for basic fields.

```
26 // calculate salaried employee's pay
27 double SalariedEmployee::earnings() const
28 {
29    return getWeeklySalary();
30
31 } // end function earnings
32
33 // return salaried employee's salary
34 double SalariedEmployee::getWeeklySalary() const
35 {
36    return weeklySalary;
37
38 } // end function getWeeklySalary
39
40 // print salaried employee's name
41 void SalariedEmployee::print() const
42 {
43    cout << "\nsalaried employee: ";
44    Employee::print();  // code reuse
45
46 } // end function print
```

Must implement pure virtual functions.

Outline

salaried.cpp
(2 of 2)


```
1  // Fig. 10.27: hourly.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 );
15    double getWage() const;
16
17    void setHours( double );
18    double getHours() const;
19
20    virtual double earnings() const;
21    virtual void print() const;
22
23 private:
24    double wage;   // wage per hour
25    double hours;  // hours worked for week
26
27 }; // end class HourlyEmployee
28
29 #endif // HOURLY_H
```


Outline

hourly.h (1 of 1)

Outline

hourly.cpp (1 of 3)

```
1  // Fig. 10.28: hourly.cpp
2  // HourlyEmployee class member-function definitions.
3  #include <iostream>
4
5  using std::cout;
6
7  #include "hourly.h"
8
9  // constructor for class HourlyEmployee
10 HourlyEmployee::HourlyEmployee( const string &first,
11    const string &last, const string &socialSecurityNumber,
12    double hourlyWage, double hoursWorked )
13    : Employee( first, last, socialSecurityNumber )
14 {
15    setWage( hourlyWage );
16    setHours( hoursWorked );
17
18 } // end HourlyEmployee constructor
19
20 // set hourly employee's wage
21 void HourlyEmployee::setWage( double wageAmount )
22 {
23    wage = wageAmount < 0.0 ? 0.0 : wageAmount;
24
25 } // end function setWage
```


Outline

hourly.cpp (2 of 3)

```
26
27 // set hourly employee's hours worked
28 void HourlyEmployee::setHours( double hoursWorked )
29 {
30    hours = ( hoursWorked >= 0.0 && hoursWorked <= 168.0 ) ?
31       hoursWorked : 0.0;
32
33 } // end function setHours
34
35 // return hours worked
36 double HourlyEmployee::getHours() const
37 {
38    return hours;
39
40 } // end function getHours
41
42 // return wage
43 double HourlyEmployee::getWage() const
44 {
45    return wage;
46
47 } // end function getWage
48
```

Outline

hourly.cpp (3 of 3)

```
49 // get hourly employee's pay
50 double HourlyEmployee::earnings() const
51 {
52    if ( hours <= 40 )  // no overtime
53       return wage * hours;
54    else               // overtime is paid at wage * 1.5
55       return 40 * wage + ( hours - 40 ) * wage * 1.5;
56
57 } // end function earnings
58
59 // print hourly employee's information
60 void HourlyEmployee::print() const
61 {
62    cout << "\nhourly employee: ";
63    Employee::print();  // code reuse
64
65 } // end function print
```


Outline

commission.h
(1 of 1)

```
1  // Fig. 10.29: commission.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
12       const string &, double = 0.0, double = 0.0 );
13
14    void setCommissionRate( double );
15    double getCommissionRate() const;
16
17    void setGrossSales( double );
18    double getGrossSales() const;
19
20    virtual double earnings() const;
21    virtual void print() const;
22
23 private:
24    double grossSales;      // gross weekly sales
25    double commissionRate;  // commission percentage
26
27 }; // end class CommissionEmployee
28
29 #endif  // COMMISSION_H
```

Must set rate and sales.

Outline

commission.cpp
(1 of 3)

```
1  // Fig. 10.30: commission.cpp
2  // CommissionEmployee class member-function definitions.
3  #include <iostream>
4
5  using std::cout;
6
7  #include "commission.h"  // Commission class
8
9  // CommissionEmployee constructor
10 CommissionEmployee::CommissionEmployee( const string &first,
11    const string &last, const string &socialSecurityNumber,
12    double grossWeeklySales, double percent )
13    : Employee( first, last, socialSecurityNumber )
14 {
15    setGrossSales( grossWeeklySales );
16    setCommissionRate( percent );
17
18 } // end CommissionEmployee constructor
19
20 // return commission employee's rate
21 double CommissionEmployee::getCommissionRate() const
22 {
23     return commissionRate;
24
25 } // end function getCommissionRate
```


Outline

commission.cpp
(2 of 3)

```
26
27 // return commission employee's gross sales amount
28 double CommissionEmployee::getGrossSales() const
29 {
30     return grossSales;
31
32 } // end function getGrossSales
33
34 // set commission employee's weekly base salary
35 void CommissionEmployee::setGrossSales( double sales )
36 {
37    grossSales = sales < 0.0 ? 0.0 : sales;
38
39 } // end function setGrossSales
40
41 // set commission employee's commission
42 void CommissionEmployee::setCommissionRate( double rate )
43 {
44     commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
45
46 } // end function setCommissionRate
47
```

Outline

commission.cpp
(3 of 3)

```
48 // calculate commission employee's earnings
49 double CommissionEmployee::earnings() const
50 {
51    return getCommissionRate() * getGrossSales();
52
53 } // end function earnings
54
55 // print commission employee's name
56 void CommissionEmployee::print() const
57 {
58    cout << "\ncommission employee: ";
59    Employee::print();  // code reuse
60
61 } // end function print
```


Outline

s.h (1 of 1)

Inherits from **CommissionEmployee** (and from **Employee** indirectly).

```
1  // Fig. 10.31: baseplus.h
2  // BasePlusCommissionEmployee class derived from Em
3  #ifndef BASEPLUS_H
4  #define BASEPLUS_H
5
6  #include "commission.h"  // Employee 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 );
15    double getBaseSalary() const;
16
17    virtual double earnings() const;
18    virtual void print() const;
19
20 private:
21    double baseSalary;       // base salary per week
22
23 }; // end class BasePlusCommissionEmployee
24
25 #endif // BASEPLUS_H
```

Outline

baseplus.cpp
(1 of 2)

```
1  // Fig. 10.32: baseplus.cpp
2  // BasePlusCommissionEmployee member-function definitions.
3  #include <iostream>
4
5  using std::cout;
6
7  #include "baseplus.h"
8
9  // constructor for class BasePlusCommissionEmployee
10 BasePlusCommissionEmployee::BasePlusCommissionEmployee(
11    const string &first, const string &last,
12    const string &socialSecurityNumber,
13    double grossSalesAmount, double rate,
14    double baseSalaryAmount )
15    : CommissionEmployee( first, last, socialSecurityNumber,
16      grossSalesAmount, rate )
17 {
18    setBaseSalary( baseSalaryAmount );
19
20 } // end BasePlusCommissionEmployee constructor
21
22 // set base-salaried commission employee's wage
23 void BasePlusCommissionEmployee::setBaseSalary( double salary )
24 {
25    baseSalary = salary < 0.0 ? 0.0 : salary;
26
27 } // end function setBaseSalary
```


Outline

baseplus.cpp
(2 of 2)

```
28
29 // return base-salaried commission employee's base salary
30 double BasePlusCommissionEmployee::getBaseSalary() const
31 {
32     return baseSalary;
33
34 } // end function getBaseSalary
35
36 // return base-salaried commission employee's earnings
37 double BasePlusCommissionEmployee::earnings() const
38 {
39     return getBaseSalary() + CommissionEmployee::earnings();
40
41 } // end function earnings
42
43 // print base-salaried commission employee's name
44 void BasePlusCommissionEmployee::print() const
45 {
46    cout << "\nbase-salaried commission employee: ";
47    Employee::print();  // code reuse
48
49 } // end function print
```

Outline

fig10_33.cpp
(1 of 4)

```
1  // Fig. 10.33: fig10_33.cpp
2  // Driver for Employee hierarchy.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7  using std::fixed;
8
9  #include <iomanip>
10
11 using std::setprecision;
12
13 #include <vector>
14
15 using std::vector;
16
17 #include <typeinfo>
18
19 #include "employee.h"    // Employee base class
20 #include "salaried.h"    // SalariedEmployee class
21 #include "commission.h"  // CommissionEmployee class
22 #include "baseplus.h"    // BasePlusCommissionEmployee class
23 #include "hourly.h"      // HourlyEmployee class
24
```


Outline

fig10_33.cpp
(2 of 4)

```
25 int main()
26 {
27    // set floating-point output formatting
28    cout << fixed << setprecision( 2 );
29
30    // create vector employees
31    vector < Employee * > employees( 4 );
32
33    // initialize vector with Employees
34    employees[ 0 ] = new SalariedEmployee( "John", "Smith",
35       "111-11-1111", 800.00 );
36    employees[ 1 ] = new CommissionEmployee( "Sue", "Jones",
37       "222-22-2222", 10000, .06 );
38    employees[ 2 ] = new BasePlusCommissionEmployee( "Bob",
39       "Lewis", "333-33-3333", 300, 5000, .04 );
40    employees[ 3 ] = new HourlyEmployee( "Karen", "Price",
41       "444-44-4444", 16.75, 40 );
42
```

```
43    // generically process each element i
44    for ( int i = 0; i < employees.size()
45
46       // output employee information
47       employees[ i ]->print();
48
49       // downcast pointer
50       BasePlusCommissionEmployee *commissionPtr =
51          dynamic_cast < BasePlusCommissionEmployee * >
52             ( employees[ i ] );
53
54       // determine whether element points to base-salaried
55       // commission employee
56       if ( commissionPtr != 0 ) {
57          cout << "old base salary: $"
58               << commissionPtr->getBaseSalary() << endl;
59          commissionPtr->setBaseSalary(
60             1.10 * commissionPtr->getBaseSalary() );
61          cout << "new base salary with 10% increase is: $"
62               << commissionPtr->getBaseSalary() << endl;
63
64       } // end if
65
66       cout << "earned $" << employees[ i ]->earnings() << endl;
67
68    } // end for
69
```

Use downcasting to cast the employee object into a **`BasePlusCommissionEmployee`**. If it points to the correct type of object, the pointer is non-zero. This way, we can give a raise to only **`BasePlusCommissionEmployee`**s.

Outline

fig10_33.cpp
(4 of 4)

```
70    // release memory held by vector employees
71    for ( int j = 0; j < employees.size(); j++ ) {
72
73       // output class name
74       cout << "\ndeleting object of "
75            << typeid( *employees[ j ] ).name();
76
77       delete employees[ j ];
78
79    } // end for
80
81    cout << endl;
82
83    return 0;
84
85 } // end main
```

`typeid` returns a **`type_info`** object. This object contains information about the operand, including its name.

Outline

fig10_33.cpp
output (1 of 1)

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

commission employee: Sue Jones
social security number: 222-22-2222
earned $600.00

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
old base salary: $300.00
new base salary with 10% increase is: $330.00
earned $530.00

hourly employee: Karen Price
social security number: 444-44-4444
earned $670.00

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